Module Compiler™
Reference Manual

Version W-2004.12, December 2004

Comments?
Send comments on the documentation by going to http://solvnet.synopsys.com, then clicking "Enter a Call to the Support Center."

SYNOPSYS®

Copyright Notice and Proprietary Information

Right to Copy Documentation

The license agreement with Synopsys permits licensee to make copies of the documentation for its internal use only. Each copy shall include all copyrights, trademarks, service marks, and proprietary rights notices, if any. Licensee must assign sequential numbers to all copies. These copies shall contain the following legend on the cover page:

> "This document is duplicated with the permission of Synopsys, Inc., for the exclusive use of _____ and its employees. This is copy number _____."

Destination Control Statement

All technical data contained in this publication is subject to the export control laws of the United States of America. Disclosure to nationals of other countries contrary to United States law is prohibited. It is the reader's responsibility to determine the applicable regulations and to comply with them.

Disclaimer

SYNOPSYS, INC., AND ITS LICENSORS MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

Registered Trademarks (®)

Synopsys, AMPS, Arcadia, C Level Design, C2HDL, C2V, C2VHDL, Cadabra, Calaveras Algorithm, CATS, CSim, Design Compiler, DesignPower, DesignWare, EPIC, Formality, HSPICE, Hypermodel, I, iN-Phase, in-Sync, Leda, MAST, Meta, Meta-Software, ModelAccess, ModelTools, NanoSim, OpenVera, PathMill, Photolynx, Physical Compiler, PowerMill, PrimeTime, RailMill, Raphael, RapidScript, Saber, SiVL, SNUG, SolvNet, Stream Driven Simulator, Superlog, System Compiler, Testify, TetraMAX, TimeMill, TMA, VCS, Vera, and Virtual Stepper are registered trademarks of Synopsys, Inc.

Trademarks (™)

abraCAD, abraMAP, Active Parasitics, AFGen, Apollo, Apollo II, Apollo-DPII, Apollo-GA, ApolloGAII, Astro, Astro-Rail, Astro-Xtalk, Aurora, AvanTestchip, AvanWaves, BCView, Behavioral Compiler, BOA, BRT, Cedar, ChipPlanner, Circuit Analysis, Columbia, Columbia-CE, Comet 3D, Cosmos, CosmosEnterprise, CosmosLE, CosmosScope, CosmosSE, Cyclelink, Davinci, DC Expert, DC Expert *Plus*, DC Professional, DC Ultra, DC Ultra Plus, Design Advisor, Design Analyzer, Design Vision, DesignerHDL, DesignTime, DFM-Workbench, DFT Compiler, Direct RTL, Direct Silicon Access, Discovery, DW8051, DWPCI, Dynamic Model Switcher, Dynamic-Macromodeling, ECL Compiler, ECO Compiler, EDAnavigator, Encore, Encore PQ, Evaccess, ExpressModel, Floorplan Manager, Formal Model Checker, FoundryModel, FPGA Compiler II, FPGA *Express*, Frame Compiler, Galaxy, Gatran, HDL Advisor, HDL Compiler, Hercules, Hercules-Explorer, Hercules-II, Hierarchical Optimization Technology, High Performance Option, HotPlace, HSPICE-Link, i-Virtual Stepper, iN-Tandem, Integrator, Interactive Waveform Viewer, Jupiter, Jupiter-DP, JupiterXT, JupiterXT-ASIC, JVXtreme, Liberty, Libra-Passport, Libra-Visa, Library Compiler, Magellan, Mars, Mars-Rail, Mars-Xtalk, Medici, Metacapture, Metacircuit, Metamanager, Metamixsim, Milkyway, ModelSource, Module Compiler, MS-3200, MS-3400, Nova Product Family, Nova-ExploreRTL, Nova-Trans, Nova-VeriLint, Nova-VHDLlint, Optimum Silicon, Orion_ec, Parasitic View, Passport, Planet, Planet-PL, Planet-RTL, Polaris, Polaris-CBS, Polaris-MT, Power Compiler, PowerCODE, PowerGate, ProFPGA, ProGen, Prospector, Protocol Compiler, PSMGen, Raphael-NES, RoadRunner, RTL Analyzer, Saturn, ScanBand, Schematic Compiler, Scirocco, Scirocco-i, Shadow Debugger, Silicon Blueprint, Silicon Early Access, SinglePass-SoC, Smart Extraction, SmartLicense, SmartModel Library, Softwire, Source-Level Design, Star, Star-DC, Star-MS, Star-MTB, Star-Power, Star-Rail, Star-RC, Star-RCXT, Star-Sim, Star-SimXT, Star-Time, Star-XP, SWIFT, Taurus, Taurus-Device, Taurus-Layout, Taurus-Lithography, Taurus-Process, Taurus-Topography, Taurus-Visual, Taurus-Workbench, TimeSlice, TimeTracker, Timing Annotator, TopoPlace, TopoRoute, Trace-On-Demand, True-Hspice, TSUPREM-4, TymeWare, VCS Express, VCSi, Venus, Verification Portal, VFormal, VHDL Compiler, VHDL System Simulator, VirSim, and VMC are trademarks of Synopsys, Inc.

Service Marks (SM)

MAP-in, SVP Café, and TAP-in are service marks of Synopsys, Inc.

SystemC is a trademark of the Open SystemC Initiative and is used under license.
ARM and AMBA are registered trademarks of ARM Limited.
All other product or company names may be trademarks of their respective owners.

Printed in the U.S.A.

Document Order Number: 31848-000 WB
Module Compiler Reference Manual, version W-2004.12

Contents

What's New in This Release xii

About This Manual xii

Customer Support xv

1. Language Syntax

Nonoperators 1-2

[ ] Bit Range Brackets and Substring Brackets 1-3

#define, #include, #ifdef, #endif 1-5

function, endfunction 1-8

function call 1-11

global 1-15

if/else 1-15

input 1-17

integer, integer constant 1-19

module, endmodule 1-22

output 1-25

replicate, repl 1-28

signed, unsigned 1-30
wire 1-32
Operators 1-34
~ Bitwise Negation Signal Operator 1-36
+, –, * Sum-Based Signal Operators 1-37
<<, >>, <<<, >>> Shift and Rotate Signal Operators 1-40
!=, ==, >=, >, <, <= Comparison Signal Operators. 1-42
&, |, ^ AND, OR, and XOR Signal Operators 1-44
{ } (substitute) 1-45
?: Conditional Operator 1-47

2. Functions

Hardware Functions 2-2
accum 2-4
AccPM 2-6
alup 2-9
bitrev 2-15
buffer 2-15
cat 2-17
compGE 2-18
count 2-19
crc 2-22
csconvertAddsub 2-24
csconvertSelectopOneHot 2-26
csconvertTruncate 2-27
decode 2-30

demux 2-31
divide 2-33
DW_add_fp 2-36
DW_cmp_fp 2-36
DW_div_fp 2-36
DW_flt2i_fp 2-37
DW_i2flt_fp 2-37
DW_mult_fp 2-38
fir 2-38
gfxBit 2-41
gfxBlend 2-44
gfxLogicop 2-48
gfxShift 2-50
isolate 2-52
join 2-53
LZ 2-54
mac, maccs 2-56
mag 2-59
max2, maxmin, min2 2-61
multp 2-63
norm, norm1 2-65
registers: eqreg, eqreg1, eqreg2, ensreg, preg, sreg 2-69
ResolveLatency, ResolveLatencyLoop 2-73
sat, sati 2-76
SetLatency 2-77
sgnmult 2-79

shiftlr 2-81
Supporting Functions 2-83
abs 2-84
critpath, critmode, enablepath, disablepath 2-84
csconvert 2-88
directive 2-90
error, fatal, warning, info 2-92
fnArgs 2-93
hidelat 2-94
formatStr 2-95
isEven, isOdd 2-96
is2expN, is2expNmin1 2-97
max, min 2-99
param 2-100
showgroup 2-101
string, string constant 2-103
width, log2 2-106

3. Directives

archopt 3-6
archtype 3-6
async_preset and async_clear 3-7
carrysave 3-7
clock 3-8
col 3-8

dcopt 3-8
delay 3-9
delstate 3-9
direxit 3-9
fadelay 3-10
fatype 3-10
group 3-12
indelay 3-13
inload 3-13
intround 3-14
logopt 3-14
maxtreedepth 3-14
modname 3-15
multtype 3-15
muxtype 3-17
outdelay 3-18
outload 3-18
physical 3-19
physicalfunction 3-19
pipeline 3-19
pipeslack 3-20

pipestall 3-20
round 3-20
rpgen 3-21
row 3-21
scan 3-21
selectop 3-22
use_for_size_only 3-22

4. Generic Functions

Required and Optional Cells 4-2
Generic Functions Listed Alphabetically 4-8

5. Environment Variables and dc_shell Commands

Environment Variables 5-2
Command for Starting the Module Compiler GUI 5-20
dc_shell Commands for Running Module Compiler 5-20

6. Error Messages

List of Error Messages 6-2

7. Performance Data

2-Input Adders 7-3
Incrementors 7-8

Comparators 7-10
Multipliers 7-12
Decoders 7-15
Multiplexers 7-16
Shifters and Rotators 7-18
Normalized 7-20
Equality Compare 7-21
AND, OR, XOR Logic 7-22
Symmetric Pipelined FIR Filters 7-23
Asymmetric Pipelined FIR Filters 7-25
Index

Preface

The *Module Compiler Reference Manual* introduces the basic principles of the Module Compiler tool from Synopsys, version W-2004.12, and describes how Module Compiler facilitates the task of ASIC datapath design. This preface includes the following sections:

- What's New in This Release
- About This Manual
- Customer Support

What's New in This Release

Information about new features, enhancements, and changes; known problems and limitations; and resolved Synopsys Technical Action Requests (STARs) is available in the *Module Compiler Release Notes* in SolvNet.

To see the *Module Compiler Release Notes*,

1. Go to the Synopsys Web page at http://www.synopsys.com and click SolvNet.
2. If prompted, enter your user name and password. (If you do not have a Synopsys user name and password, follow the instructions to register with SolvNet.)
3. Click Release Notes in the Main Navigation section (on the left), move your pointer to Module Compiler, then choose the release you want in the menu that appears.

About This Manual

This section provides information on related publications, online resources, conventions, and customer support.

Audience

This manual is for designers who are familiar with

- VHDL or Verilog
- The UNIX operating system

- The X Window System
- The basic concepts of synthesis and simulation

Prior knowledge of computer-aided engineering (CAE) tools, ASIC design flow, and digital hardware structures is helpful.

Related Publications

For additional information about Module Compiler, see

- *Module Compiler User Guide*
- Synopsys Online Documentation (SOLD), which is included with the software for CD users or is available to download through the Synopsys Electronic Software Transfer (EST) system
- Documentation on the Web, which is available through SolvNet at http://solvnet.synopsys.com
- The Synopsys MediaDocs Shop, from which you can order printed copies of Synopsys documents, at http://mediadocs.synopsys.com

You might also want to refer to the documentation for the following related Synopsys products:

- Physical Compiler
- Design Compiler
- Power Compiler

Conventions

The following conventions are used in Synopsys documentation.

Convention	Description
`Courier`	Indicates command syntax.
`Courier italic`	Indicates a user-defined value in Synopsys syntax, such as *`object_name`*. (A user-defined value that is not Synopsys syntax, such as a user-defined value in a Verilog or VHDL statement, is indicated by regular text font italic.)
`Courier bold`	Indicates user input—text you type verbatim—in Synopsys syntax and examples. (User input that is not Synopsys syntax, such as a user name or password you enter in a GUI, is indicated by regular text font bold.)
[ ]	Denotes optional parameters, such as *`pin1`* [*`pin2 ... pinN`*]
|	Indicates a choice among alternatives, such as `low | medium | high` (This example indicates that you can enter one of three possible values for an option: low, medium, or high.)
_	Connects terms that are read as a single term by the system, such as `set_annotated_delay`
Control-c	Indicates a keyboard combination, such as holding down the Control key and pressing c.
\	Indicates a continuation of a command line.
/	Indicates levels of directory structure.
Edit > Copy	Indicates a path to a menu command, such as opening the Edit menu and choosing Copy.

Customer Support

Customer support is available through SolvNet online customer support and through contacting the Synopsys Technical Support Center.

Accessing SolvNet

SolvNet includes an electronic knowledge base of technical articles and answers to frequently asked questions about Synopsys tools. SolvNet also gives you access to a wide range of Synopsys online services including software downloads, documentation on the Web, and “Enter a Call to the Support Center.”

To access SolvNet,

1. Go to the SolvNet Web page at http://solvnet.synopsys.com.
2. If prompted, enter your user name and password. (If you do not have a Synopsys user name and password, follow the instructions to register with SolvNet.)

If you need help using SolvNet, click HELP in the top-right menu bar or in the footer..

Contacting the Synopsys Technical Support Center

If you have problems, questions, or suggestions, you can contact the Synopsys Technical Support Center in the following ways:

- Open a call to your local support center from the Web by going to http://solvnet.synopsys.com (Synopsys user name and password required), then clicking “Enter a Call to the Support Center.”
- Send an e-mail message to your local support center.
  - E-mail support_center@synopsys.com from within North America.
  - Find other local support center e-mail addresses at http://www.synopsys.com/support/support_ctr.
- Telephone your local support center.
  - Call (800) 245-8005 from within the continental United States.
  - Call (650) 584-4200 from Canada.
  - Find other local support center telephone numbers at http://www.synopsys.com/support/support_ctr.

1

Language Syntax

This chapter defines the language syntax available in the Module Compiler tool and provides examples of its usage. It consists of the following sections:

- Nonoperators
- Operators

Nonoperators

This section defines the nonoperators language syntax available in Module Compiler and provides examples of the usage of the following nonoperators:

- [ ] Bit Range Brackets and Substring Brackets
- #define, #include, #ifdef, #endif
- function, endfunction
- function call
- global
- if/else
- input
- integer, integer constant
- module, endmodule
- output
- replicate, repl
- signed, unsigned
- wire

[ ] Bit Range Brackets and Substring Brackets

[ ] Bit range brackets for signals

[ ] Substring brackets for strings

Syntax

```
<signal> [<integer value> : 0]

<signal|string> [<integer value>]

<signal|string> [<high value>: <low value>]
```

Arguments

<integer value>

Any expression that evaluates to an integer

<high value>

Any expression that evaluates to an integer

<low value>

Any expression that evaluates to an integer

Directives

None

Description

You can use the bracket pair for three purposes: to declare the size of a signal variable, to select a substring of a string variable, or to specify a range of bits in a previously declared signal variable.

To declare the size of a signal variable, use the first or second syntax form. Since all signals must start and go up from bit zero, the first syntax form requires that the second or lower value be zero. In this first syntax form, all signals are declared as [n : 0], where *n* is an integer value greater than or equal to 0. For the second syntax form, all signals are declared as [n], where *n* is the size of the signal. This second syntax is shorthand for the first syntax; therefore, it could be written instead as [n - 1 : 0], where *n* is an integer value greater than 0. Use either of these forms in all input, output, and wire declarations.

To specify a range of bits of a previously declared signal variable, use the second or third syntax style. Use these forms wherever a previously declared signal is allowed, except when the signal is in carry-save format.

The range of bits selected is from the high index through the low index. If only one index is specified, the bit range selects only that one bit. The range must not exceed the size of the variable. In the second syntax form, the index must be greater than or equal to zero; in the third syntax, the low value must be greater than or equal to zero and the high value must be greater than or equal to the low value.

To select a substring of a string variable, use the second or third syntax style. The range of characters selected is from the high index through the low index.

The first (leftmost) character has an index of 0, and the last (rightmost) character has an index of strlen(var) –1. If only one index is specified, only one character is selected. The range must not exceed the number of characters in the string.

A bit range is considered signed if (and only if) the base signal is signed and the bit range includes the sign bit.

A bit range cannot appear on the left side of an assignment expression. Similarly, a bit range cannot be used as the output of a function. Carry-save signals cannot be referenced by a bit range.

Example 1-1 Bit Range Limitations

```
wire unsigned [4:0] B, A;
B [4:3] = A [2:1]; // this is not allowed.
```

The results shown in Example 1-1 can be achieved by:

```
B = cat (A[2:1],B[2:0]);
```

Example 1-2 Bit Range Signal and Substring Brackets

```
Z = A[5] ? A[4:0]:A [10:6]; // select bits 4-0 or 10-6
                                // depending on bit 5

string test = "the quick brown fox";

string t1 = test[4:5] + test[2] + test[2] + test[14];
// t1="queen"
```

#define, #include, #ifdef, #endif

#define

Defines a preprocessor macro

#include

Includes another Module Compiler file

#ifdef

Preprocessor condition

#endif

Preprocessor condition

Syntax

```
#define <name> [(<arg list>)] definition

#include "<file>"

#ifdef <preprocessor condition>

#endif <preprocessor condition>
```

Directives

None

Description

Module Compiler supports the use of the standard C language preprocessor (cpp) constructs. These constructs are resolved in an initial preprocessor pass before any other parsing takes place.

The `#define` function provides a name for a text block. The preprocessor substitutes the text block wherever it encounters the macro name. The preprocessor also performs argument substitution during this process.

The `#include` function includes the named file in the current position in the input stream.

The `#ifdef` function conditionally invokes the preprocessor. It must be used with the `#endif` function.

The `#endif` function conditionally invokes the preprocessor. It must be used with the `#ifdef` function.

Note:

To prevent collisions between the Verilog-style constant syntax, such as 20'h 1f374, and C language preprocessor string literals, such as 'this is a string', string literals must be delimited by double quotes and never by single quotes.

For information about C language preprocessor constructs, see the cpp man page documentation. This documentation is a standard part of most UNIX systems.

Example 1-3 #define, #include

```
#define MAX(a, b) if (a>b) {a} else {b}
//if a > b then insert a wherever MAX(a, b) is used
//else insert b
#include "myFunc.mc"
//include the file myFunc.mc in the current input stream
```

Example 1-4 Using #ifdef

```
#define DEBUG 1
module test (Z, X, Y1, Y2, in, out);
  string name = "testCell";
  directive(modname = name);

  #ifdef DEBUG 1
  info("name of the output cell is: ", name, "\n");
  #endif

  integer in, out;
  output [out - 2] Z;
  input [in - 2] X, Y1, Y2;

  #ifdef DEBUG 1
  info ("input width is: ", in, "\t output width is: ", out,
  "\n");
  #endif

  Z = X *(Y1 + Y2);
endmodule
```

function, endfunction

function

Starts the definition of a function

endfunction

Ends the definition of a function

Syntax

```
[integer | string] function <name> ([<variable list>]);
...
endfunction
```

Arguments

<name>

<function name>

<variable list>

<variable> [, <variable>]*

Directives

None

Description

A function is the Module Compiler equivalent of a software procedure. It is a block of Module Compiler Language code that is abstracted into a named entity. By referring to its name, you can instantiate copies of this code.

The code that calls this entity can itself be a similar entity. Thus, it is possible to have hierarchies of functions in which each higher-level function is built with calls to lower-level functions or building blocks.

These functions are abstract entities: They are pieces of code that have no meaning outside Module Compiler processing. When this processing is complete, all function calls are resolved and the result is a flat netlist.

Integer functions can be used in integer expressions, and string functions can be used in string expressions.

The `function` statement declares the beginning of the function block and the function interface. The function interface consists of arguments that can be any of the supported data types but that especially include signal inputs and outputs.

The `endfunction` indicates the end of the function description. Note that `endfunction` is not terminated by the usual semicolon.

The function contents can include any statements other than a module or a function declaration. The function can declare new variables and make other function calls. The function can modify its arguments according to the following rules:

- Integers and strings are passed in by value; any modifications of these values in this function have no effect on the caller.
- Signals are passed in by reference. If the signal is modified, these changes are visible in the caller when the function returns. Input signals cannot be modified.

Output signals can be modified in either of the following ways:

- The function can assign width and format to an output. The caller can declare the output without a width in this case. If the caller declares the output with a width that does not match the one assigned in the function, a conversion is made through a temporary variable.

- The function can assign a value to the output signal by using an expression or by passing it to another function. This output must be previously unassigned.

Fixed argument lists as well as variable argument lists are supported.

Module Compiler implements strict type checking between expected and actual arguments for fixed argument list functions. This requires that the number as well as the type of arguments match. A notable exception is that an integer value can be passed in wherever a signal input is expected.

Variable-length-signal argument lists are identified by the `VAR` keyword after the function name. Integer arguments with default values at the end of the list need not be supplied by the caller.

In addition, a variable-length list of signal arguments can be created by use of `replicate` or `repl` in conjunction with the `fnArgs` function, which returns the number of arguments supplied to the function.

For additional information, see “function call” on page 1-11.”

Example 1-5 function, endfunction

```
function test(Z, X, Y);// function named test has 3 arguments
  ...
  input unsigned [8] X;
  // 8-bits wide, unsigned function input
  ...
endfunction


function test(Z, X, U, V, n);
  // function named test has 5 arguments
  ...
  input [8] X;
  // 8-bits wide, unsigned by default function input

  input [8] U, V;
  // 8-bit wide, unsigned function inputs U, V

  integer n; // integer argument
  ...
endfunction
```

---

function call

Function name used to make a function call

Syntax

```
[string | integer] <function name> [<instance name>]
([<arg list>]);
```

Arguments

<name>

<function name>

<instance name>

<name of a call identifier>

<arg list>

<expression> [, <expression>]*

Description

A function call implements the reference to the collection of Module Compiler code that is a function. Because functions accept arguments, it is possible to execute the same body of code on different operands.

Arguments to the function must match in number and type. Function arguments are matched from the left. Thus, if the calling statement is

```
someFunction(A, B, n, str)
```

and the function declaration is

```
function someFunction(X, Y, i, char)
```

then A is mapped to X, B is mapped to Y, n is mapped to i, and str is mapped to char. The types of n and i must match. Similarly, the types of str and char must match. The one exception to this rule is that an integer constant can be passed in where a signal input is expected.

For example,

```
someFunction(A, 15, n, str)
```

A function call causes the body of the function to be replicated inside the parent. The replication takes place at the location of the function call. All arguments are substituted and all nonsignal variables and expressions are resolved before the replication. When the replication is complete, processing resumes immediately after the calling expression or statement in the caller.

Replicating the contents of the function in the context of the caller requires that all variables created in the function have nonconflicting names in the caller. This is achieved by the renaming of all signals that were declared with wire.

The new name is usually of the form

```
<prefix>_<name>_<integer count>_
```

<prefix>_

Is a concatenation of one of more names

<name>_

Is the name as declared; in Example 1-6, it is Z

<integer count>_

Is an optional count used to ensure uniqueness

Note that these names always end in the underscore character (_).

Serial function-calling styles are supported:

```
<arg1> = someFunction (<arg2>, <arg3>, ...)

someFunction(<arg1>, <arg2>, <arg3>, ...)
```

These two formats are equivalent. In either case, if the function creates a wire, the name of the wire is prefixed by the name of <arg1>. The name of this signal can be suffixed by an integer value to ensure uniqueness.

If <arg1> is not the first output, the name of the first output is used instead, provided that this output is declared before any wire is declared.

```
<arg1> = someFunction <instance name> (<arg2>, <arg3>, ...)
someFunction <instance name> (<arg1>, <arg2>, <arg3>, ...)
```

These two formats are equivalent. If the function creates a signal, the name of the signal is prefixed by <instance name>. The name of this signal can be suffixed by an integer value to ensure uniqueness.

```
<variable> = <expression>
```

In this case, <expression> can include one or more arbitrarily nested function calls, such as

```
X = Y + someFunction (<arg2>, <arg3>, ...)) +
otherFunction(A, someFunction(<arg2>, <arg3>, ...));
```

Note that in each case, the embedded call to someFunction does not specify <arg1>. Each occurrence of such an embedded function call leads to the creation of a temporary variable used to represent the function output. This effectively reduces the expression to

```
<temp variable> = someFunction (<arg2>, <arg3>, ...);
<temp variable 2> = someFunction (<arg2>, <arg3>, ...);
<temp variable 3> = otherFunction (A, <temp variable2> ...);
X = Y + <temp variable> + <temp variable 3> + ...
```

This requires that the functions in question do not rely on the caller to assign attributes to the outputs.

Functions are not required to be defined before they are called. Function calls can be made from files other than the one containing the function definition.

Example 1-6 function call

```
input X, Y;
wire [8] Z;
someFunction(Z, X, Y);
// a call to someFunction which uses
// the width of Z as declared here

input X, Y;
wire [8] Z; // same as above except that "call_1"
someFunction call_1 (Z, X, Y);//the call identifier

input X, Y;
wire Z;
// no width declared; width is determined by someFunction
Z = someFunction(X, Y); // note difference in calling style

input X, Y;
wire [8] Z;
Z = someFunction(X, Y); //identical to first example above
```

global

Declares a wire, integer, or string as global

See “integer, integer constant” on page 1-19 and “string, string constant” on page 2-103.

if/else

Implements conditional text inclusion

Syntax

```
if (<expression>) {<block>}

if (<expression>) {<block>} else {<block>}
```

Arguments

<expression>

An expression that evaluates to an integer

<block>

An arbitrary block of code

Directives

None

Description

The if/else construct provides flow control in the Module Compiler description and allows the input to be conditionally modified before being further processed.

This construct can appear inside other constructs, statements, and expressions, including function and module interface declarations.

If/else, replicate, repl, and substitution can be intermixed and nested. There are some limitations, which are described below. See also “replicate, repl” on page 1-28 and “{ } (substitute)” on page 1-45.

The if/else construct provides conditional code inclusion. The condition expression is any expression that results in a true or false integer value.

If the expression is true, the block following the condition is inserted into the input stream; otherwise, the block following the else construct is inserted into the input stream. A block is checked for syntactical correctness only if it is inserted into the input stream.

The conditional expression in if/else cannot contain any flow control constructs or substitutions.

Example 1-7 if/else

```
if (formatStr(X) == "signed")
    {wire [width(X)] signed Y;}
else
    {wire [width(X)] unsigned Y;}
    // create a wire same width and format as X;
```

---

input

Declares a signal input for a function or a module

Syntax

```
input [fb] [unsigned | signed] [<range>] <variable list>;
```

Arguments

<range>

[<integer expression> : 0]

<variable list>

<signal> [, <signal>]*

Directives

inload

For module inputs, maximum input load, in 0.1 standard load

indelay

For module inputs, arrival time, in ps

Description

All variables in Module Compiler Language must be explicitly declared before they are referenced. The input construct declares one or more signal variables, which are treated as the inputs to the function or the module containing the input statement.

If the input statement appears in a module, it must have a width specification. The format specification is optional and is assumed to be unsigned. Inputs to a module can be further characterized by the directives listed above.

If the input statement appears in a function, it can have a width or format specification. If a width and format are specified that do not match those of the signal passed to the function, a temporary variable is generated to perform the conversion. If the width and format are not specified, no restrictions are placed on the function caller and no temporary variables are generated.

Unlike other signal variables, module inputs are considered assigned to and can therefore be used immediately on the right side in an expression. By the same token, module inputs cannot be assigned to. For proper operation, inputs to a function must have been assigned to by the caller before the function call.

Function inputs can be declared as points at which loops can be broken. You can do this by including the `fb` keyword after input, as indicated in the syntax description. The Module Compiler library functions have this attribute specified already.

Example 1-8 Declares a Signal Input

```
module top(Z, X, Y);
  ...
input unsigned [8] X; //8-bit-wide unsigned module input

module top(Z, X, U, V);
  ...
input [8] X;
// 8-bit-wide (unsigned by default) module input
input [8] U, V;
// 8-bit-wide unsigned module inputs, U and V

function someFunction(Z, X, Y);
  ...
input X; //input for a function, with
         //attributes declared by caller

module top (Z, X, Y);
  ...
input [1] X; //1-bit-wide module input
```

---

integer, integer constant

integer

Declares an integer variable

integer constant

Declares an integer constant

Syntax

```
integer [global] <variable list>;
```

Arguments

<variable list>

<assigned variable> [, <assigned variable>]*

<assigned variable>

<variable> [= <expression>]

<expression>

Integer expression

Syntax

```
<constant> [<size>] ' <format> <value>
```

Arguments

<size>

Number of bits required (0–1,024)

<format>

h | d | o | b for hex, decimal, octal, or binary

Directives

None

Description

All variables in Module Compiler Language must be explicitly declared before they are referenced. The integer construct declares and optionally initializes an integer variable. Unlike integer variables, integer constants can be used anywhere without being declared.

Integer constants and integer variables are both treated as 32-bit numbers by default. If these numbers are used in a signal expression, then the bit-width for the constant is automatically computed.

For example,

```
Z = X + Y + 5; // 5 is a 3-bit quantity
```

Integer variables and integer constants can be made to take on values larger than 32 bits by use of format specifiers as shown above. Large integers support a limited number of operators, such as + and –, but do not support other operators, such as /.

```
integer x = 'h123456789abcde; //x is a big integer
integer y = x + 16; // Allowed
integer z = x / 16; // Not Allowed
integer a = 160;
integer b;
b = b + a / 16; // Allowed
```

For a list of supported integer operators, see Table 1-2 on page 1-35.

Global integer variables can be accessed throughout the entire design (the module and all functions).

Table 1-1 lists the integer functions that are supported. These functions return integer values. For more information, see the entry for each of these functions throughout this chapter.

Table 1-1 Integer Functions

Integer function	Description
abs(<integer x>)	Returns the absolute value of x
fnArgs	Returns the number of arguments passed to a function
log2(<integer x>)	Returns the ceiling of the log base 2 of x
max(<integer x, integer y>)	Returns the maximum of x and y

Table 1-1 Integer Functions (Continued)

Integer function	Description
min(<integer x, integer y>)	Returns the minimum of x and y
width(<signal x>)	Returns the width of a signal or constant x

Example 1-9 Declares an Integer

```
integer x, y, z; // declare three integers
integer xx = 15; // declare and initialize an integer
integer xxx = 64 'h 10;
// declare a 64-bit integer and initialize it to hex 10
```

Example 1-10 Integer Functions

```
wire [8] X;
integer x, y, z; // declare three integers
integer xx = min(x,max(y,z));
//declare and initialize an integer
integer xxx = width(X); //xxx=8
```

module, endmodule

module

Begins the definition of a module

endmodule

Ends the definition of a module

Syntax

```
module <name> ([<variable list>]);

...

endmodule
```

Arguments

<name>

<module name>

<variable list>

<variable> [, <variable>]*

Directive

modname

The name of the cell in the output

Description

The `module` construct describes the design to be synthesized. The description begins with `module` and ends with `endmodule`. Note that the `endmodule` statement is not terminated by the usual semicolon.

The `module` statement describes the interface to the cell, whereas the description specifies the contents of the cell. The `module` statement declares the name of the module and the arguments (inputs, outputs, parameters) to the module.

Nonsignal arguments are resolved by Module Compiler. Signal arguments are used to construct the interface to the synthesized cell. Simulation models that are generated by Module Compiler are given signal declarations in the same order as the signals in the module parameter list.

Typically, the output cell and the various output files that are unique to a module are named after the module. However, it is possible to override this by using the modname directive.

If the module declaration includes a parameter (such as an integer or a string), an appropriate value must be provided, with a construct such as <n> = <value> for all parameters without default values or for parameter values that are used in the argument list. You can specify the value from the command line or by using the Module Compiler graphical user interface (GUI).

Example 1-11 module and endmodule

```
module top(Z, X, Y);
//a module, top, with three arguments, and output cell, top
  ...
  input unsigned [8] X;
  //8-bit-wide unsigned module input
  ...
endmodule


module top(Z, X, U, V, n);
//a module named top, with five arguments
  ...
  input [8] X;
  // 8-bit-wide (unsigned by default) module input
  input [8] U, V; //8-bit-wide unsigned module inputs U, V
  integer n;  // integer value
  ...
  directive (modname = "mcCell");
  //name the output cell mcCell
  ...
endmodule
```

output

Declares a signal output for a function or module

Syntax

```
output [unsigned | signed] [<range>] <variable list>;
```

Arguments

<range>

[<integer expression> : 0]

<variable list>

<variable> [, <variable>]*

<variable>

<signal> | <signal> = <expression>n

Directives

outdelay

For module outputs, the external delay, in ps

outload

For module outputs, the load in 0.1 standard load

Description

All variables in Module Compiler Language must be explicitly declared before they are referenced. The `output` construct declares one or more signal variables, which are treated as the outputs of the function or the module containing the `output` statement.

If the `output` statement appears in a module, it can have a width specification. If a width is not specified, a function that declares its output width must be connected to the output. The format specification is optional and is assumed to be unsigned. Outputs to a module can be further characterized by the directives listed above.

If the `output` statement appears in a function, the width specification can be omitted and the attributes of the signal are assumed to be specified by the caller of this function or by some other function called from this function.

If the width specification is not omitted, the attributes of the signal can be undefined in the caller of this function. If the caller defines the attributes differently, precedence is given to the declaration inside the function and then a temporary operand is created to convert between the two widths.

Output signals, like wires, are considered not assigned to. Consequently, output signals must be assigned to before they can be used in the right side of an expression.

Example 1-12 output Statement

```
module top(Z, X, Y);
  ...
  output unsigned [8] Z;
  // 8-bit-wide unsigned module output
  output [1] X;   // 1-bit-wide module input


module top(Z, X, Y, U, V);
  ...
  output [8] Z;
  // 8-bit-wide (unsigned by default) module output
  output signed [8] X, Y;
  // 8-bit-wide signed module outputs X,Y


function someFunction(Z, X, Y);
  ...
  output Z;
  // function output with caller-declared attributes


function someFunction(Z, X, Y);
...
output signed [8] Z;
// output, width 8, format signed.
// attributes of Z, if caller defined,
// function must match caller declaration

output [1] A = X ^ Y, B = X + Y;
// declaration and assignment
```

replicate, repl

Implements a loop

Syntax

```
replicate (<statement>; <expression>; <statement>
           [; <separator> ]) { <block> }
repl (<var>, <limit> [, <separator>]) {<block>}
```

Arguments

<expression>

Expression that evaluates to an integer

<block>

Arbitrary block of code

<statement>

Arbitrary statement

<separator>

Quoted string

<var>

Name of the loop variable

<limit>

Integer expression

Directives

None

Description

The `replicate` and `repl` constructs provide flow control in the Module Compiler description and allow the input to be conditionally modified before being further processed. These constructs can appear inside other constructs, statements, and expressions, including function and module interface declarations.

`If/else`, `replicate`, `repl`, and `substitution` can be intermixed and nested. However, there are some limitations, which are described in this section. See also “if/else” on page 1-15 and “{ } (substitute)” on page 1-45.

The `replicate` construct is similar to for-loop in the C/C++ language. This construct replicates the given block of code into the input stream, as long as the loop condition holds. It is customary to use the construct with an initial statement, which initializes a loop variable, and an update statement, which updates the loop variable in each iteration.

Use `replicate` to generate comma-separated variable lists. As shown in the following example, these lists can result in a trailing comma. This does not usually cause a problem, because such lists are valid in most contexts.

Avoid using `replicate` with a dangling operator, such as, A{i} +. This problem is generally solved by use of a null terminator. For example,

```
Z = replicate (i = 0; i < n; i = i + 1) { X{i} + } 0;
```

generates

```
Z = X0 + X1 + X2 + ... + 0
```

You can use the separator string to avoid extra trailing characters. Replicated blocks are separated by this string. No separator string leads the first block or trails the last block.

Like `if/else`, the `replicate` construct can be used anywhere in a module or function, and it can be nested with other flow control constructs. However, the start, update, and loop conditions are required to be free of flow control constructs.

The `repl` construct is a short form of `replicate`. The arguments are separated by commas. The first argument is the name of the `loop` variable, the second argument is the replication limit, and the third optional argument is the separator string.

The `loop` variable does not need to be predeclared. It is an implicitly created integer variable initialized to 0 and then incremented at the start of each iteration. The scope of this variable is strictly local to the loop.

Example 1-13 replicate and repl

```
replicate (integer i = 0; i < n; i = i + 1) { wire [8] X{i}; }
//create n signal variables, named X0, X1, X2, ...

wire [8] replicate (integer i = 0; i < n; i = i + 1; "," ) { X{i}};
//same as above

wire [8] replicate (i = 0; i < n; i = i + 1) { X{i+8}, };
//same as above, except use a previously declared variable, i,
//and name the variables X8, X9, X10, etc.

wire [8] repl (i, n, ",") { X{i+8} }; //same as above
```

signed, unsigned

signed

Declares the format of a signal to be signed

unsigned

Declares the format of a signal to be unsigned, which is the default

Syntax

```
<declaration> [unsigned | signed] <variable list>;
```

Argument

<declaration>

Wire, input, or output

Directives

None

Description

The signed and unsigned operands are used to declare the format of a signal when the signal is declared. Signed and unsigned formats up to 1,024 bits are supported for operands.

All signed operands are represented with a 2's-complement representation: The sign bit has negative significance, and all other bits have positive significance.

Unsigned numbers are represented in standard binary. The value of signed numbers is

$$-b_{n-1}2^{n-1} + \sum_{i=0}^{n-2} b_i 2^i$$

The value of unsigned numbers is

$$\sum_{i=0}^{n-1} b_i 2^i$$

Bit ranges that include the sign bit of a signal are considered signed. Constants are considered signed if they are less than 0.

Example 1-14 Signed and Unsigned Signal Declarations

```
wire signed [8] X;
input unsigned [11] input_signal;
```

---

wire

Declares a signal variable that is local to a function or module

Syntax

```
wire [global] [unsigned | signed] [<range>] <variable list>;
```

Arguments

<range>

[<integer expression> : 0]

<variable list>

<variable> [, <variable>]*

<variable>

<signal> | <signal> = <expression>

Directives

None

Description

All variables in Module Compiler Language must be explicitly declared before they are referenced. The `wire` construct declares one or more signal variables. These variables are local to the module or the function containing the `wire` statement, unless `wire` is followed by the `global` keyword.

If a format is not specified in a wire declaration, the wire is assumed to be unsigned. If a width for a wire is not specified, the wire is assumed to be undefined. In typical usage, the attributes are then assigned by a function that treats this wire as its output.

For example:

```
wire Z; // predeclare the wire
someFunction(Z, X, Y); // can assign a width/format to Z
```

A function can assign attributes to a fully defined wire as well, but in such a case, the new attributes are required to match the old definition or a temporary variable is generated to convert between the different widths and formats. For example,

```
wire [6] Z; // predeclare the wire
someFunction(Z, X, Y); // can reassign width [6] to Z
```

Wires declared inside modules retain the name used in the declaration. Wires declared inside functions appear in the output with an automatically generated name. For information about the naming scheme, see “function call” on page 1-11.

A wire declared as global can be used anywhere in the design. Local signals take precedence over global signals with the same name. Global wires should be used sparingly.

Example 1-15 wire

```
wire unsigned [8] X;
// 8-bit-wide unsigned signal

wire global [8] X;
// 8-bit-wide global signal unsigned by default

wire [8] X, Y, Z;
// 8-bit-wide unsigned signals named X, Y, Z

wire X; // signal X with undefined width
wire [1] X; // 1-bit-wide unsigned signal

wire [1] A = X ^ Y,B = X + Y;
// declaration and assignment
```

Operators

This section defines the operator language syntax available in Module Compiler and provides usage examples. It includes the following operators:

- ~ Bitwise Negation Signal Operator
- +, –, * Sum-Based Signal Operators
- <<, >>, <<<, >>> Shift and Rotate Signal Operators
- !=, ==, >=, >, <, <= Comparison Signal Operators
- &, |, ^ AND, OR, and XOR Signal Operators
- { } (substitute)
- ?: Conditional Operator

Table 1-2 lists the integer operators that are supported. The meaning and the precedence of these operators are defined by the C programming language and by the order they appear in the table.

Table 1-2 Integer Operators

Integer operator	Name
( )	Expression grouping
–	(Unary) negate
+	(Unary) add
!	Logical negation
~	Bitwise negation
*	Multiply
/	Divide
%	Modulus
+	Add
–	(Binary) minus
>>	Right shift
>>>	Rotate right shift
<<	Left shift
<<<	Rotate left shift
==	Equality compare
!=	Inequality
<	Less than
>	Greater than

Table 1-2 Integer Operators (Continued)

Integer operator	Name
<=	Less than or equal to
>=	Greater than or equal to
&	Bitwise AND
^	Bitwise XOR
|	Bitwise OR
&&	Logical AND
||	Logical OR
? :	Conditional

~ Bitwise Negation Signal Operator

Performs unary signal inversion

Syntax

```
<signal> = ~ <expression>
```

Arguments

<expression>

Expression using any supported operators

Directives

None

Description

The ~ operator performs signal inversion. The corresponding operator for integers has the usual interpretation (see “integer, integer constant” on page 1-19). You can mix integers and signals in the same expression.

Every bit in the binary representation of the result is the inverse of what was in the (converted) expression.

Example 1-16 Invert Signal Operator

```
wire [1] A = ~B;      // invert B
wire [1] Z = A | ~B; // A or invert B
```

+, –, * Sum-Based Signal Operators

\+ Binary or unary addition of signals

– Binary or unary subtraction of signals

* Multiplication of signals

Syntax

```
<signal> = [<unary op>] <expression> [<binary op>
<expression>] *
```

Arguments

<unary op>

+ or –

<binary op>

+ or –or *

<expression>

Expression using any supported operators

Directives

carrysave

Carry-save mode

direxT

Forces direct sign extension

fadelay

Final adder delay goal, in ps

fatype

Type of adder to build (aofcla; cla; clsa; csa; fastcla; ripple; ripple_alt; or auto, the default)

intround

Internal rounding control

maxtreedepth

Maximum Wallace tree depth

multtype

Multiplier type

round

Rounds result to given position

Description

The +, –, and * operators implement the add, subtract, and multiply operations for signals, respectively. Corresponding operators for integers have the usual interpretation (see “integer, integer constant” on page 1-19). You can mix integers and signals in the same expression.

Any number of operands can be combined in an expression. Constants and bit ranges can be used on the right side of the expression. Signal operands used with a + or –operator can be left-shifted by an integer value. The second signal operator in a multiply can be left-shifted by an integer value.

Caution!

Left-shifting the first operator in a multiply can lead to extra hardware.

Example 1-17 Sum-Based Signal Operators

```
wire [8] Y = B - C + D; // uses one adder

wire [8] Y1 = B * (C + D)
```

<<, >>, <<<, >>> Shift and Rotate Signal Operators

<< Left-shift operator for signals

>> Right-shift operator for signals

<<< Left-rotate operator for signals

>>> Right-rotate operator for signals

Syntax

```
<signal> = <input operand> <operator> <shift operand>
```

Arguments

<operator>

<< | >> | <<< | >>>

<shift operand>

<signal> | <integer> | <constant>

<input operand>

<signal> | <integer> | <constant>

Directive

selectop

Type of select (shift input) optimization: msb, lsb, auto

Description

The << operator implements left shift. The >> operator implements arithmetic or logical right shift for signed and unsigned inputs, respectively. The quantity on the left side of the operator is the input, and the quantity on the right side is the shift value.

If both the input and the shift value are integers, integer shift takes place. Otherwise, signal shift takes place, as described in this section.

If the shift value is a signal, a shifter is implemented in hardware.

For the shift operators, the input data is always directly sign extended if the output is wider than the input. If the output is narrower than the input, the full-precision output is simply truncated after shifting. The least significant bits (LSBs) are returned, and unused most significant bits (MSBs) are truncated. A logical left shift of a signed input is implemented by first converting the input operand to unsigned.

The <<< operator performs a left rotation; the >>> operator performs a right rotation. Unlike the shift operators, where bits shifted out the ends are lost, bits shifted out of one end of the rotators wrap around to the other end.

Negative constant shift values cause a shift in the opposite direction. If the data input is constant, the logic optimizer is relied upon to reduce the area.

The `selectop` attribute controls the ordering of select signals for shifters, rotators, and MUX-based multiplexers. This attribute has the following settings:

- When `selectop` is set to msb, the select inputs are ordered from MSB to LSB, where the delay from the LSB is the least and the delay from the MSB is the greatest.
- When `selectop` is set to lsb, the ordering is reversed, with the delay from the LSB being the greatest.
- When `selectop` is set to auto, Module Compiler orders the select inputs to minimize the delay from the select inputs to the output, based on the select input arrival times.

Example 1-18 Shift and Rotate

```
wire [8] A = B << C;
//Left-shift B by C

wire [8] Y = B << (C >> 2);
```

!=, ==, >=, >, <, <= Comparison Signal Operators

== Equality operator

!= Inequality operator

>= Greater-than-or-equal-to operator

> Greater-than operator

< Less-than operator

<= Less-than-or-equal-to operator

Syntax

```
<signal> = <expression> <operator> <expression>
```

Arguments

<expression>

Expression using any supported operators

<operator>

!= or == or >= or > or < or <=

Directives

direxT

Direct sign extension for intermediate sum

intround

Internal rounding control

round

Rounds result to given position

Description

The == operator tests for the equality of two signal expressions. The result is always a 1-bit unsigned value that is 1 if the two inputs are equal and is 0 otherwise. The two inputs can be of different widths and formats and are considered equal only if they represent the same integer with due regard to the sign.

The >, <, >=, and <= operators are based on subtraction. The result is computed with the sign bit of the result of a subtraction involving the left and right sides. The size of the intermediate result is computed to be large enough to avoid any overflows.

Example 1-19 Comparison Signal Operator

```
wire [1] A = B == C;
wire [1] X = (A + B) > (C + D); // uses a single adder
wire [8] Y = B >= C ? B : C; // max (B, C)
```

&, |, ^ AND, OR, and XOR Signal Operators

| Bitwise OR or unary OR

& Bitwise AND or unary AND

^ Bitwise XOR or unary XOR

Syntax: Bitwise

```
<signal> = <expression> <operator> <expression> [<operator>
<expression>] *
```

Syntax: Unary

```
<signal> = <operator> <expression>
```

Arguments

<operator>

& or | or ^

<expression>

Expression using any supported operators

Directives

None

Description

The ^, |, and & operators implement the XOR, OR, and AND operations for signals. Corresponding operators for integers have the usual interpretation (see “integer, integer constant” on page 1-19). You can mix integers and signals in the same expression.

When you use one of these signal operators in a binary fashion, a bitwise operation is performed. Reduction to a single bit occurs when you use the operator in a unary fashion.

Any number of operands can be combined in an expression. Constants and bit ranges can be used on the right side of the expression. Signal operands can be inverted.

The signal variable receiving the result must be large enough to hold all required MSBs. Unused MSBs are truncated.

Example 1-20 AND, OR, and XOR

```
wire [8] Y = B | C | ~D;

wire [8] Z = B | C | ~D & E;

wire [1] Z = &B
//AND reduction of B
```

{ } (substitute)

Computes a value and substitutes it into the input stream

Syntax

```
{<expression>}
```

Arguments

<expression>

Expression that evaluates to a string or an integer

Directives

None

Description

The { } construct (substitute) provides a mechanism for evaluating an expression and substituting its value into the input stream. For example, an input of X{5 + 10} becomes X15. The expression inside the { } construct can be arbitrarily complex.

Example 1-21 Substitute

```
integer i;
string s;
...
Z{s}{i}={s}{i}>>SH;
// If the value of s is 'Dec' and the value of i is 13,
// this statement becomes ZDec13 = Dec13>>SH;
```

?: Conditional Operator

Used to implement conditional expressions using multiplexers

Syntax

```
<signal> = <expression> ?  <expression> : <expression>
[: <expression>] *
```

Arguments

<expression>

Expression using any supported operators

Directives

muxtype

Multiplexer type

Description

The ?: operator is used to implement conditional expressions using multiplexers. Specify the signal used in selecting the signal to the left of ? and the list of signals to be selected to the right of ?. Separate the signals to be selected with colons (:).

Module Compiler selects these signals from right to left as the select input value increases from 0 to n–1. For example, if the select signal is 0, Module Compiler selects the rightmost signal. If the select signal is 1, Module Compiler selects the second-rightmost signal, and so on.

You can use an *n*-bit-wide select signal to multiplex 2^n signals. It is not necessary to specify the entire range of inputs: If only *m* inputs are specified, then the top m+1 to 2n inputs are not connected and are treated as don't care values for the purpose of the optimization.

It is also possible to create holes by specifying don’t care values, using the constant 'h x for the corresponding input. Use the don’t care values when possible to decrease the area required to implement the multiplexer.

Example 1-22 Conditional Operator

```
directive (muxtype="mux");
wire [8]  V = select[2] ? B : C[15:8] : D : A<<3;
wire [16] W = ~(select ? B : A);
  directive (muxtype="andor");
wire [16] X = select ? B : A;
  directive (muxtype="tristate");
wire [16] Y = select ? B : A;
wire [16] Z = select ? B : A : 'h x;
//don't care what is output when select=0
```

2

Functions

This chapter defines functions available in Module Compiler and provides examples of their usage. It includes lists that present the functions that

- Generate datapath gates (hardware functions)
- Perform conditional synthesis and do not generate gates (supporting functions)

This chapter consists of the following sections:

- Hardware Functions
- Supporting Functions

Hardware Functions

The following functions generate datapath gates:

- accum
- AccPM
- alup
- bitrev
- buffer
- cat
- compGE
- count
- crc
- csconvertAddsub
- csconvertSelectopOneHot
- csconvertTruncate
- decode
- demux
- divide
- DW_add_fp
- DW_cmp_fp
- DW_div_fp

- DW_flt2i_fp
- DW_i2flt_fp
- DW_mult_fp
- fir
- gfxBit
- gfxBlend
- gfxLogicop
- gfxShift
- isolate
- join
- LZ
- mac, maccs
- mag
- max2, maxmin, min2
- multp
- norm, norm1
- registers: eqreg, eqreg1, eqreg2, ensreg, preg, sreg
- ResolveLatency, ResolveLatencyLoop
- sat, sati
- SetLatency

- sgnmult
- shiftlr

---

accum

Accumulator signal function

Syntax

```
accum (<output Z>, <input X>, <input R>, <input S>);
```

Arguments

<output Z>

Output signal; width/format declared by the caller

<input X>

Input signal named X, the value to accumulate

<input R>

Input signal named R; width 1 bit, active-low reset

<input S>

Input signal named S, accumulator reset value

Directives

carrysave

Must be set to off

fatype

Type of final adder to build (aofcla; cla; clsa; csa; fastcla; ripple; ripple_alt; or auto, the default)

intround

Must be set to 0

pipeline

Must be set to off

round

Must be set to 0

Description

The `accum` function builds an accumulator. The signal to be accumulated is specified by the X input, and the result is specified by the Z output.

The width of the accumulator is the same as the width of the output. Use the input signal named S as the start value of the accumulator. Use the input signal named R to reset the accumulator value to the S value plus the input.

The exact function implemented by `accum` is

$Z(t) = R(t-1) \; ? \; Z(t-1) + X(t-1) : S(t-1) + X(t-1)$

The `accum` function cannot be embedded in other expressions, because this embedding requires the generation of a temporary variable for the output, Z. This temporary variable cannot be automatically generated, because its size needs to be supplied by the designer.

Example

```
module accum_test (OUT, IN, RESET, START);
  integer w1 = 28, w2 = 29, w3 = 32;
  input unsigned [w1-2] IN;
  input unsigned [1] RESET;
  input unsigned [w2-2] START;
  output unsigned [w3-2] OUT;
  accum (OUT, IN, RESET, START);
endmodule
```

AccPM

Extended-precision digital signal processing (DSP) multiplier-accumulator, Z = C + / −X * Y

Syntax

```
AccPM (<output Z>, <input C>, <input X>, <input Y>,
       <input ADD>, <input XS>, <input YS>);
```

Arguments

<output Z>

Output signal; width/format declared by the caller

<input C>

Accumulator input signal

<input X>

The multiplier X input

<input Y>

The multiplier Y input

<input ADD>

1-bit input signal; 0 means -x * y, 1 means +x * y

<input XS>

1-bit input signal; 0 means X is unsigned, 1 means X is signed

<input YS>

1-bit input signal; 0 means Y is unsigned, 1 means Y is signed

Directives

carrysave

Controls the carry-save format

fatype

Type of final adder to build (aofcla; cla; clsa; csa; fastcla; ripple; ripple_alt; or auto, the default)

intround

Internal rounding control

pipeline

Enables/disables automatic pipelining

round

Rounds result at given position

Description

`AccPM` is the core of a multiplier-accumulator commonly used in extended-precision digital-signal-processing applications.

Positive and negative products can be accumulated, and the format of the X and Y inputs can be varied dynamically through XS and YS. This function produces a purely combinational core without registers. The exact function implemented by `AccPM` is

Z = ADD ? C + X * Y : C –X * Y

Consider the extended-precision problem of multiplying two signed numbers, XI and YI, each 32 bits, using a 16-bit-wide `AccPm` block.

Note that XI = XU << 16 + XL and YI = YU << 16 + YL. XU and YU are the signed upper 16 bits of X and Y, respectively. XL and YL are the unsigned lower 16 bits of X and Y, respectively. Table 2-1 sums the four partial products and sets XS and YS.

Table 2-1 AccPM

Product	XS	YS
YU*XU	1	1
YU*XL	0	1
YL*XU	1	0
YL*XL	0	0

`AccPM` cannot be embedded in other expressions, because this embedding requires the generation of a temporary variable for the output, Z. This temporary variable cannot be automatically generated, because its size needs to be supplied by the designer.

Example

```
integer Ywidth,Xwidth,Cwidth;
input [Xwidth-2] X;
input [Ywidth-2] Y;
input signed [Cwidth-2] C;
input [1] Add,XS,YS;
output [Cwidth-2] Z;
Z=AccPM(C,X,Y,Add,XS,YS);
```

alup

Provides a programmable arithmetic logic unit (ALU) with up to 16 instructions; the width and instruction set are also programmable.

Syntax

```
alup (<output Z>, <input A>, <input B>, <input DI>,
<output DO>, <input CI>, <input INST>, <output FLAGS>,
<input FirstCyc>, <integer instMask>);
```

Arguments

<output Z>

Primary output with width determined by caller

<input A, input B>

Primary inputs, w=max(width(A),width(B))

<input DI>

Dividend input (divide only)

<output DO>

Dividend output (divide only), width = w

<input CI>

1-bit-wide carry input signal, ignored if FirstCyc is high

<input INST>

4-bit-wide instruction input, selects the current operation (see Table 2-2). If INST is set to the value of a masked instruction, Z is undefined.

Table 2-2 alup Instruction Input

INST	Function	Z
0	IncA	A+1
1	DecA	A-1
2	Add	A+B
3	Sub	A-B
4	Abs	abs(A)
5	NegA	-A
6	Divide	A/B
7	NegB	-B
8	And	A&B
9	Or	A|B
10	XOR	A^B
11	InvB	~B
12	PassA	A
13	InvA	~A
14	Zero	0
15	One	1

<output FLAGS>

4-bit-wide condition code (flag) output (see Table 2-3). If a flag bit is masked, its value should be treated as undefined.

Table 2-3 alup Flag Output

Bit	Function	Description
0	OvfFlag	High indicates that an overflow occurred in an arithmetic operation
1	CarryFlag	High indicates that a carry-out occurred
2	ZeroFlag	High indicates that a zero result occurred
3	Reserved	Do not use; for future enhancement

<input FirstCyc>

1-bit input. FirstCyc should be high during the first cycle of extended-precision operations. Also, FirstCyc should be high during the division operation. CI is ignored if FirstCyc is high.

<integer instMask>

Integer bit mask. Sets each bit to mask the corresponding instruction or output flag (see Table 2-4). instMast = 0 provides full functionality; instMask = –1 provides none.

Table 2-4 alup Integer Bit Mask

Bit	Function	Description
0	IncA	Clear to enable A + 1 instruction
1	DecA	Clear to enable A –1 instruction
2	Add	Clear to enable A + B instruction
3	Sub	Clear to enable A – B instruction
4	Abs	Clear to enable abs(A) instruction
5	NegA	Clear to enable –A instruction

Table 2-4 alup Integer Bit Mask (Continued)

Bit	Function	Description
6	Divide	Clear to enable division instruction
7	NegB	Clear to enable -B instruction
8	And	Clear to enable A&B instruction
9	Or	Clear to enable A|B instruction
10	XOR	Clear to enable A^B instruction
11	InvB	Clear to enable ~B instruction
12	PassA	Clear to enable A instruction
13	InvA	Clear to enable ~A instruction
14	Zero	Clear to enable zero instruction
15	One	Clear to enable one instruction
16	OvfFlag	Clear to enable overflow-flag output
17	CarryFlag	Clear to enable carry-flag output
18	ZeroFlag	Clear to enable zero-flag output

Directive

fatype

Type of final adder to build (aofcla; cla; clsa; csa; fastcla; ripple; ripple_alt; or auto, the default)

Description

The `alup` function provides a programmable arithmetic logic unit (ALU) with as many as 16 instructions. During normal operation, the A and B inputs compute the output Z, with INST as the number of the instruction to execute. Several flags are available for indicating special conditions.

You can disable unneeded instructions and flag bits individually during compilation to reduce the delay and the area of the ALU. When an instruction or a flag bit is disabled, the output Z or the flag bit, respectively, is undefined when the instruction is selected.

You can implement a four-quadrant division algorithm serially, with one quotient bit computed in each step. At the beginning of the operation, the dividend is supplied on the DI inputs (B is set to 0 if DI> = 0 or −1 if DI < 0) and the divisor on the input A. After the first cycle, the output Z is fed back to the input B and the output DO is fed back to the DI output.

After all iterations (width cycles) are completed, Z is the remainder and DO is the quotient. Two final processing steps are required to complete the division. If the divisor is negative, the quotient must be negated. If the remainder is less than 0, the absolute value of the divisor must be added to the remainder.

Extended-precision operation is supported for the IncA, DecA, Add, Sub, NegA, and NegB instructions. The data is processed from LSB to MSB width bits at a time. In the first cycle, the signal FirstCyc is set high and the CI input is ignored.

On subsequent cycles, FirstCyc is set low and the FLAG[1] output (carry-out) is fed back to the CI input. For other operations or when using single-precision operation, you can set FirstCyc to high, and CI

is ignored. It is also possible to set FirstCyc low for all operations, in which case you must supply the correct carry input value on the CI input.

Three flags are available. When the OVF flag is set high, it indicates that a 2's-complement overflow occurred. If the inputs and outputs are unsigned, this bit has no meaning.

The carry flag is connected to the carry output of the adder. It indicates overflow for unsigned operations and provides the CI input value for extended-precision operations.

The OVF and carry flags are undefined for logical instructions. To force these flags to 0, the outputs should be ANDed with ~INST[3]. The zero flag is valid for all instructions and indicates that Z is 0.

Example

```
module mc_alu (Z, A, B, DI, DO, CI, INST, FLAGS, FirstCycle);
   integer      Bits     = 3;
   input  [Bits] A,B,DI;
   output [Bits] Z, DO;
   input  [1] CI, FirstCycle;
   input  [4] INST;
   output [4] FLAGS;
   string   mult_type = "nonbooth";
   string   adder_type = "fastcla";
   directive (fatype= adder_type);
   directive (multtype = mult_type);
   alup (Z, A, B, DI, DO, CI, INST, FLAGS, FirstCycle,0);
endmodule
```

bitrev

Bit reversal function

Syntax

```
bitrev (<output Z>, <input X>);
```

Arguments

<output Z>

Output signal; width same as the input, always unsigned

<input X>

Input signal

Directives

None

Description

The `bitrev` function forms the output by reversing the bits from the input. The MSB of the input becomes the LSB of the output and vice versa. No hardware is generated by this function.

Example

```
input [8] A;
wire X = bitrev(A[4]);// X=A[0],A[1],A[2],A[3]
```

buffer

Signal function for setting buffer depth

Syntax

```
buffer (<output Z>, <integer n>);
```

Arguments

<output Z>

Output signal; width/format declared by the caller

<integer n>

Integer representing buffer depth

Directives

None

Description

The `buffer` function builds a buffer tree to drive heavy loads.

Note:

Because `buffer` modifies the given signal directly instead of producing a new signal that is the modified form of the input, the unbuffered output is not available when `buffer` is used.

If a signal is buffered more than once, the greatest buffer depth takes precedence. Because `buffer` does not produce a new signal that is the modified form of the input, it is not meaningful to embed `buffer` in an expression.

The maximum buffer depth is 5.

Example

```
input [8] A;
wire [8] ANC;
someFunction(ANC, A); // ANC is the output of someFunction
buffer (ANC,2); // ANC has buffer depth
```

cat

Signal concatenation functions

Syntax

```
cat (<output Z>, <input X1>, <input X2>, ..., <input Xn>);
```

Arguments

<output Z>

Output signal; width = sum of widths, format = format of X1

<input X1>

First input signal

<input X2>

Second input signal

<input Xn>

Last input signal

Directives

None

Description

The `cat` function accepts n input signals and concatenates them to create the output, Z. The number of inputs must be greater than 0 (n > 0).

The first input, X1, forms the MSB of the output, and the last input, Xn, forms the LSB. You can use constants to create “holes” between signals. It is not possible to create overlaps.

The output format and bit range are determined from the input signals and must not be declared differently by the caller. The width of the output is equal to the sum of the widths of the inputs, and the output is signed if the first input, X1, is signed. Otherwise, the output is unsigned.

Example

```
input [8] A,B,C;
wire X; // bitrev determines width and format
X=cat(C[2],A[4],B[6:5]);
// means X=C[1],C[0],A[3],A[2],A[1],A[0],B[6],B[5]
```

---

compGE

Performs a numerical comparison of two inputs to form a 2-bit output, indicating if the first input is greater than or equal to the second input.

Syntax

```
compGE(<output Z>, <input X>, <input Y>);

Z=compGE(X,Y);
```

Arguments

<input X, input Y>

Input signals; can be any width and format

<output Z>

2-bit output signal; Z[1] is true if X > Y, Z[0] is true if X == Y

Directives

fatype

Type of final adder to build (aofcla; cla; clsa; csa; fastcla; ripple; ripple_alt; or auto, the default)

Description

The `compGE` function performs a numerical comparison of two inputs, X and Y, to form the 2-bit output Z.

It is not possible to merge arithmetic expressions or compare carry-save values with `compGE`, so you might want to use the inferred comparators when either of these situations arises.

If Z[0] is set, the two inputs are equal. If Z[1] is set, X is greater than Y. The numerical value of X is the numerical value of Y, considering the width and the format of each.

Note:

It is possible for two signals to have the same bit pattern but not be equal. In the following example, the equal bit of the output, Z[0], is 0.

```
wire signed [8] X = -1;
wire [8] Y = 255;
wire [2] Z=compGE(X,Y);
// Z=(0,0), X not greater and not equal
```

If you want a bitwise comparison, make sure X and Y are unsigned.

Example

```
wire [2] Z;
compGE(Z,A,B);
```

---

count

Signal function for creating a counter

Syntax

```
count (<output Z>, <input X>, <input R>, <input S>, <integer
detectOVF>, <output OVF>);
```

Arguments

<output Z>

Output signal; width/format declared by the caller

<input X>

Counter control input

<input R>

Input signal named R; 1-bit, unsigned, active-low reset

<input S>

Input signal named S, counter reset value

<integer detectOVF>

Integer flag; detect overflow if set

<output OVF>

Output signal; set by count to be 1-bit unsigned

Directives

carrysave

Must be set to off

fatype

Type of final adder to build (aofcla; cla; clsa; csa; fastcla; ripple; ripple_alt; or auto, the default)

pipeline

Must be set to off (cannot pipeline the loop)

round

Must be set to 0

Description

The `count` function builds a simple up and down counter. By choosing the width and format of the X input, you can control whether the counter counts up, down, or up and down.

With a 1-bit unsigned X input, the counter counts up by 1 (X = 1) or holds constant (X = 0). With a 1-bit signed X input, the counter counts down by 1 (X = –1) or holds (X = 0). With a 2-bit signed X input, the counter counts by –2, –1, 0, or 1. A 1-bit reset input, R, is provided to reset the counter to the start value, S.

If `count` is called with detectOVF set to true, the OVF argument must be passed in. The counter automatically resets to the start value when overflow occurs, and OVF then contains the overflow output. The `count` function creates OVF as a 1-bit unsigned signal.

The exact function implemented by `count` is given by

Z(t) = R(t–1) ? Z(t–1) + X(t–1) : S(t–1)

If R is low, Z goes to the value of S in the next cycle. Otherwise, the input, X, is added to the previous value of the counter.

The `count` function cannot be embedded in other expressions, because this embedding requires the generation of a temporary variable for the output, Z. This temporary variable cannot be automatically generated, because its size needs to be supplied by the designer.

The function always expects Z, X, R, X and <integer detectOVF>. If <integer detectOVF> is set, it expects <output OVF>.

Example

```
input [1] R, S;//module inputs with preassigned values
wire [1] X;
wire [8] Z0, Z1, Z2;
count(Z0,X,R,S,0); //simple up counter
count(Z1,X,R,S,1,OVF);
//simple up counter, load start on overflow
wire signed [2] X;
count(Z2,X,R,S,0); //up/down counter
```

---

crc

Generates a CRC encoder/decoder

Syntax

```
crc (<output Z>, <output ERR>, <input X>, <input R>,
<input GEN>, <integer Degree>);
```

Arguments

<output Z>

Output signal; width/format declared by the caller

<output ERR>

Output signal for parity errors; width/format declared by the caller

<input X>

Input signal

<input R>

Input signal representing reset, active-low

<input GEN>

Input signal representing the polynomial

<integer Degree>

Integer specifying the degree of the generator polynomial

Directives

None

Description

Cyclic redundancy checking (CRC) is a common error-detection method. The `crc` function provides a bit-serial CRC encoder/decoder that operates with programmable or fixed generator polynomials. The degree of the generator polynomial is specified via the Degree integer.

The polynomial itself is provided on the GEN input, with each bit of GEN providing one Boolean coefficient of the polynomial. The LSB of the GEN input corresponds to the constant term coefficient of the polynomial. The generator polynomial always has a coefficient of 1 for the integer degree-th term.

Resetting the encoder means bringing the R input low for degree cycles.

During encoding, the R input is held high while the data bits are clocked in on the X input. The R input is then held low for degree cycles as the parity bits are clocked out on the Z output.

During decoding, the R input is held high while the data bits are clocked in on the X input. The R input can then be held low for degree cycles to clock the parity bits out on the Z output, or the ERR signal can be examined to test for nonzero parity. ERR is set to high when a parity error occurs.

The caller can declare the Z and ERR outputs to have any width or format.

Example

```
input [18] generator;

input [1] d, reset;

wire [1] Parity, err, Parity1, err1;

crc(Parity, err,d,reset,generator,18);
// degree 18 variable generator

integer generator1 = 18'h0a13c; // constant generator

crc(Parity1, err1,d,reset,generator1,18);
// degree 18 constant generator
```

---

csconvertAddsub

Builds an addition or subtraction circuit

Syntax

```
csconvertAddsub (<output Z>, <input A>, <input B>,
     <input S>);
```

Arguments

<output Z>

Output signal

<input A>

Input signal A: The first operand

<input B>

Input signal B: The second operand

<input S>

Input signal S: The 1-bit-select signal in binary format where
S=0, operator synthesized is +
S=1, operator synthesized is –

Directive

None

Description

The `csconvertAddsub` function builds an addition or subtraction circuit, based on S, the single-bit control signal for operations of the type Z = A +/– B, where the addition or subtraction operating is

Z = A + B, if S=0 and
Z = A – B, if S=1

Inputs A and B can be in CS or binary form, whereas the S signal is always in binary format. For the output, Z, to be in the carry-save format, you must set the `carrysave` directive to `on` in the calling function or module. The input operands, A and B, and the output operand Z can be all signed or all unsigned.

If B is an implicitly truncated operand and Z is wider than B, convert B into a binary form and pass it to `csconvertAddsub`. If A and B are both in binary form, Synopsys recommends using Z = A + sgnmult(B, S), where S is a signed 1-bit signal.

Note:

The `csconvertAddsub` function is supported only by the default Module Compiler Language parser.

Example

```
input [31:0] a,b,c;
input [0:0] s;
directive (carrysave="on");
wire [63:0] e = a*b;
wire [63:0] f= csconvertAddsub(e,c,s);
directive (carrysave="off");
output [63:0] g = f+1;
```

csconvertSelectopOneHot

Returns an operand in carry-save format

Syntax

```
csconvertSelectopOneHot (<output Z>, <parameter N>,
     <input sel>, <input data0>, <input data1>...,
     <input dataN-1>);
```

Arguments

<output Z>

Output signal in carry-save format

<parameter N>

Parameter, of type integer, that specifies the width of the select signal and size of the one-hot multiplexer

<input sel>

Input signal: the select signal with a width of N

<input data0>

Input signal: input operand when sel [0] = 1

<input data1>

Input signal: input operand when sel [1] = 1

<input dataN-1>

Input signal: input operand when sel [N-1] = 1

Directives

None

Description

The `csconvertSelectopOneHot` function returns one, out of N-input operands, in carry-save format. The selection is based on the N-bit-wide-select signal, sel, in the one-hot fashion.

The N inputs can be in binary or carry-save format, independently of one another. The returned type is in carry-save format.

If all input operands are in binary, Synopsys recommends using `selectopOneHot` instead of `csconvertSelectopOneHot`. In the `selectopOneHot` function, the output, Z, is in binary as well as carry-save format.

Note:

The `csconvertSelectopOneHot` function is supported only by the default Module Compiler Language parser.

Example

```
input [31:0] a,b,c,d;
input [0:0] c0,c1;
directive (carrysave="on");
wire [63:0] e = a*b + c;
wire [63:0] f = a + b*c;
wire [63:0] g = csconvertSelectopOneHot(2,cat(c0,c1),e,f);
directive (carrysave="off");
output [63:0] h = g - d;
```

---

csconvertTruncate

Performs truncation in carry-save or binary format

Syntax

```
csconvertTruncate (<output Z>, <input A>, <input origMsb>,
     <input newMsb>, <input newLsb>, <input origLsb>,
     <parameter formZ> );
```

Arguments

<output Z>

Output signal: truncated signal in a format based on the formZ parameter

<input A>

Input signal: original operand with a range of origMsb to origLsb

<input origMsb>

Most significant bit of input A

<input origLsb>

Least significant bit of output Z

<input newMsb>

Most significant bit of output Z

<input newLsb>

Least significant bit of output Z

<parameter formZ>

Parameter, of string type, that determines binary or carry-save format

Directive

None

Description

The `csconvertTruncate` function performs bit truncation on operand A in carry-save or binary format. There are three types of bit truncation:

- Lower-bit truncation: A[origMsb:newLsb], where origMsb>=newLsb>origLib

- Upper-bit truncation: A[newMsb:origLsb], where origMsb>newMsb>=origLsb
- Upper-and-lower-bit truncation: A[newMsb:newLsb], where origMsb>newMsb>=newLsb>origLsb

This function returns a truncated signal in carry-save format (formZ="CS"); otherwise, it returns a truncated signal in binary (formZ="binary"). There are certain restrictions, which are described below.

Note:

If an operand is in binary form, Synopsys recommends truncating it directly instead of using the `csconvertTruncate` function.

Use the following rules with the `csconvertTruncate` function:

- The value of the origMsb must be greater than or equal to newMsb, newLsb, and origLsb.

  origMsb >= newMsb >= newLsb >= origLsb.

- The operand *must* be returned in binary format when the value of Z is wider than the value of truncated A for upper-bit and upper-and-lower-bit truncation cases.

Note:

The `csconvertTruncate` function is supported only by the default Module Compiler Language parser.

Example

```
input [31:0] a,b,c;
 directive (carrysave="on");
 wire [63:0] d = a*b + c;
 wire [31:0] e = csconvertTruncate(d, 63, 63, 32, 0, "cs");
 directive (carrysave="off");
 output [31:0] f = e+1;
```

---

decode

Signal function for creating a decoder

Syntax

```
decode (<output Z>, <input X>);
```

Arguments

<output Z>

Output signal; width/format declared by the caller. The format must be unsigned.

<input X>

Input signal named X

Directives

None

Description

The `decode` function builds a decoder. A partial decoder can be built by specification of an output smaller than 2x, where x is the bit-width of the input. Behaviorally, the output of decode is Z = 1 << X. For example,

```
X =  0 Z =    1
X =  1 Z =   10
```

```
X = 10 Z =  100
X = 11 Z = 1000
```

The `decode` function cannot be embedded in other expressions, because this embedding requires the generation of a temporary variable for its output.This temporary variable cannot be automatically generated, because its size needs to be supplied by the designer.

Example

```
input [4] A;
wire [8] X; // width = bits allows the decoder to decode
     // lower 3 bits of A only
decode (X, A);
```

---

demux

Signal function for creating a demultiplexer

Syntax

```
demux ( <input X>, <input S>, <output Z1>, ..., <output Zn>);
```

Arguments

<input X>

Input signal X, representing data

<input S>

Input signal S, representing select

<output Z1>

First output signal; width/format declared by `demux` to be same as the input

<output Zn>

Last output signal; width/format declared by `demux` to be same as the input

Directives

None

Description

The `demux` function builds a 1 –n demultiplexer that converts a serial data stream (X input) into n parallel data streams (Z1, Z2, ... Zn outputs). The data conversion is controlled by the select input (S input), which cycles through integer values 0 to n –1.

The input values that arrive when the select input has a value of 0, 1, 2, ... n –1 appear on the 0, 1, 2, ... n –1 indexed output, respectively. All outputs are updated the cycle after the select input has a value of 0. The following is an example for n = 4:

```
input:  0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
select: 0 1 2 3 0 1 2 3 0 1  2  3  0  1  2  3
out0:   x x x x x 0 0 0 0 4  4  4  4  8  8  8
out1:   x x x x x 1 1 1 1 5  5  5  5  9  9  9
out2:   x x x x x 2 2 2 2 6  6  6  6 10 10 10
out3:   x x x x x 3 3 3 3 7  7  7  7 11 11 11
```

The Z1, Z2, ... Zn outputs have a width/format the same as the input, by default.

The registers associated with demultiplexing are treated as state registers; therefore, no latency increase occurs in the demultiplexer. The demultiplexer latency cannot be equalized, because the delay between the input and the earliest output change varies from 2 to n + 1 cycles.

Example

```
// 1:3 demux example
wire signed [8] D0, D1, D2;
input signed [8] A; // module input with preassigned value
input [2] S0; // module input with preassigned value
demux(A,S0,D0, D1, D2);

// 1:2 demux example
wire signed [8] B, C;
input [1] S1;
demux(A,S1,B,C);
```

---

divide

Operation: Q = X / Y + R.

Divides dividend X by divisor Y, to get quotient Q and remainder R

Syntax

```
divide (<output Q>, <input X>, <input Y>, <integer Round>,
<integer Arch>, <output R>);
```

Arguments

<output Q>

Quotient output; same width and format as X

<input X>

Unsigned dividend input; can be any width

<input Y>

Unsigned divisor; can be any width for Arch = 1 and Arch = 3, but should be less than 11 bits for Arch = 2

<integer Round>

1 for biased rounding of quotient, 0 for no rounding; this argument is valid for Arch = 1 and Arch = 2 only

<integer Arch>

1 for architecture with smaller size, 2 for faster speed, 3 for best performance

<output R>

Remainder output; same width and format as Y

Directives

fatype

Type of final adder to build (aofcla; cla; clsa; csa; fastcla; ripple; ripple_alt; or auto, the default)

multtype

Type of multiplier to build (Arch = 2 only)

Description

The `divide` function divides a dividend, X, by a divisor, Y, to get quotient Q and a remainder, R. The unsigned inputs/outputs are X, Y, Q, and R.

The widths of the X and Y inputs can be controlled independently by Xwidth and Ywidth. The widths of Q and R are Xwidth and Ywidth, respectively.

For Arch = 1 and Arch = 2, Q and R are not defined when the divisor, Y, is 0. For Arch = 3, Q is the maximum value for a given bit width. The equation X = Q * Y + R is true only when Arch = 1 and Arch = 3.

There are three architectures in the `divide` function, controlled by Arch:

- If Arch = 1, a shift-subtract divider is produced and it is generally smaller than Arch = 2. The remainder (R = X % Y) is meaningful only when Arch =1 and Round is not true.

- If Arch = 2, X is multiplied by 1 / Y. A ROM generates 1 / Y; therefore, this architecture is good for small Ywidths and is generally faster than Arch = 1. There is no remainder, R, for either rounding mode in this architecture.

  The table width for Arch = 2 is a function of X and Y and is calculated with X + Y + 1. The maximum limit is 32 bits. The Y input must be less than 11 bits.

- If Arch = 3 and Round = 0, use the fastest Module Compiler divider, a restoring divider that produces 2 quotient bits per iteration. The resulting architecture sets the quotient value to its maximum possible value if the divisor is 0.

For Arch = 1 and Arch = 2, there are two rounding modes for the quotient, Q, controlled by Round. When Round is true, Q is the nearest integer of quotient X / Y. When Round is false, the quotient, X / Y, is truncated to form Q.

Example

```
Q=divide(X,Y,1,1);
// subtraction divider, no remainder (round)

divide (Q,X,Y,0,2);
// reciprocal divider, truncate Q

divide (Q,X,Y,0,1,R);
// subtraction divider, get Q and R

divide (Q,X,Y,0,3,R);
// Arch=3 optimal divider
```

DW_add_fp

Instantiates a DesignWare floating-point adder

Description

To use this DesignWare component, a DesignWare license is required. Also, it requires the installation of the DesignWare Foundation floating-point library.

For more information about this component, use the search capability at: http://www.synopsys.com/ipdirectory.

DW_cmp_fp

Instantiates a DesignWare floating-point comparator

Description

To use this DesignWare component, a DesignWare license is required. Also, it requires the installation of the DesignWare Foundation floating-point library.

For more information about this component, use the search capability at: http://www.synopsys.com/ipdirectory.

DW_div_fp

Instantiates a DesignWare floating-point divider

Description

To use this DesignWare component, a DesignWare license is required. Also, it requires the installation of the DesignWare Foundation floating-point library.

For more information about this component, use the search capability at: http://www.synopsys.com/ipdirectory.

DW_flt2i_fp

Instantiates a DesignWare floating-point-to-integer converter

Description

To use this DesignWare component, a DesignWare license is required. Also, it requires the installation of the DesignWare Foundation floating-point library.

For more information about this component, use the search capability at: http://www.synopsys.com/ipdirectory.

DW_i2flt_fp

Instantiates a DesignWare integer-to-floating-point converter

Description

To use this DesignWare component, a DesignWare license is required. Also, it requires the installation of the DesignWare Foundation floating-point library.

For more information about this component, use the search capability at: http://www.synopsys.com/ipdirectory.

DW_mult_fp

Instantiates a DesignWare floating-point multiplier

Description

To use this DesignWare component, a DesignWare license is required. Also, it requires the installation of the DesignWare Foundation floating-point library.

For more information about this component, use the search capability at: http://www.synopsys.com/ipdirectory.

fir

Generates a finite-impulse-response (FIR) filter

Syntax

```
fir (<output Z>, <integer n >, <input X>, <input Y1>, ...,
<input Y n >);
```

Arguments

<output Z>

Output signal; width/format declared by the caller

<integer n>

Integer specifying the number of taps

<input X>

Input signal

<input Y1>

Input signal representing the first coefficient

<input Yn>

Input signal representing the last coefficient

Directives

carrysave

Carry-save mode

delstate

Must be set to 0

dirext

Forces direct sign extension

fadelay

Final adder delay goal, in ps

fatype

Type of final adder to build (aofcla; cla; clsa; csa; fastcla; ripple; ripple_alt; or auto, the default)

intround

Internal rounding control

maxtreedepth

Maximum Wallace tree depth

round

Rounds result to given position

Description

The `fir` function implements a direct-form, parallel FIR filter. The filter accepts one data input signal (X), an integer (n) representing the number of taps, and a list of coefficients (Y1, ... Yn). The data input (X) and the data inputs representing coefficients (Y1, ... Yn) can all have different widths and formats.

The filter provides an output, Z. The size of the output is declared by the caller and must be large enough to hold the result. The exact function implemented is

$$Z(t) = X(t) * Y1 + X(t-1) * Y2 + ... + X(t-n+1) * Yn$$

The `fir` function cannot be embedded in other expressions, because this embedding would require the generation of a temporary variable for the output, Z. This temporary variable cannot be automatically generated, because its size needs to be supplied by the designer.

Example

```
input [8] X, C0, C1, C2;
input [5] C3;
wire [16] Z, Z1;
fir(Z,4,X,C0,C1,C2,C3); // four-tap filter
```

gfxBit

A graphics function that merges a region of bits in the input A with input B to form the output, Z.

Syntax

```
gfxBit(<output Z>,<input A>,<input B>,<input si>,
  <input so>,<input w>,
  <input UpOpt>,<input LowOpt>);
```

Arguments

<output Z>

Output containing the selected window from A; same width as A,B; always unsigned

<input A, input B>

Data inputs; must be the same width and be a power of 2 in width

<input si>

Start position of the window in the input (0 = LSB)

<input so>

Start position of the window in the output (0 = LSB)

<input w>

Width of the window minus 1 (0 -> width=1)

<input UpOpt>

2-bit input to select option for the upper region (see Table 2-5).

Table 2-5 gfxBit Upper Region

Bit	Description
0	Fill with 0
1	Fill with B[i]
2	Sign-extend from A

Be sure to set UpOpt [1] to 0 to save the sign-extension logic.

<input LowOpt>

1-bit input to select option for the lower region (see Table 2-6)

Table 2-6 gfxBit Lower Region

Bit	Description
0	Fill with 0
1	Fill with B[i]

Constraints:

si+w+1<=width(A)

so+w+1<=width(A)

width(A)==width(B)==width(Z)==2n

width(w)==width(si)==width(so)=n

Directives

None

Description

The `gfxBit` function is a graphics function that merges a region of bits in the A input with the B input to form the output, Z. The position of the bit region in A is controlled with si and w, so that w + 1 bits starting at position si are selected from A. These bits are inserted into Z at position so.

Algorithmically, the output, Z, can be computed as the following (treating any empty bit ranges as 0):

```
UpperRegion = (UpOpt ? -A[w+si] : B[width(B)-1:w+so+1]:0)
<< (w+so+1);

Window=A[w+si:si]<<so;

LowerRegion=(LowOpt ? B[so-2] : 0);

Z=UpperRegion | Window | LowerRegion;
```

The output, Z, is undefined if the constraints described above are not met.

Example

```
input [2] UpOpt;
input [1] LowOpt;
input [3] w,si,so;
input [8] A,B;
output [8] C;
C=gfxBit(A,B,si,so,w,UpOpt,LowOpt);
```

Let

A = (A7,A6,A5,A4,A3,A2,A1,A0)

B = (B7,B6,B5,B4,B3,B2,B1,B0)

The example results are shown in Table 2-7.

Table 2-7 gfxBit Example Results

si	so	w	UpOpt	LowOpt	C
0	0	3	0	0	(0, 0, 0, 0, A3,A2,A1,A0)
1	0	3	0	0	(0, 0, 0, 0, A4,A3,A2,A1)
0	1	3	0	0	(0, 0, 0, A3,A2,A1,A0,0)
2	3	1	1	1	(B7,B6,B5,A3,A2,B2,B1,B0)
2	3	1	2	1	(A3,A3,A3,A3,A2,B2,B1,B0)
2	3	1	2	0	(A3,A3,A3,A3,A2,0, 0, 0)

gfxBlend

Graphics alpha blender, or linear interpolator, programmable blend of two input pixels to form the output pixel

$Z = X\alpha + Y(1 - \alpha)$

Syntax

```
gfxBlend(<output Z>,<input X>,<input Y>,
         <input alpha>,<input alpha1>);

gfxBlend2(<output Z>,<input X>,<input Y>,
          <input alpha>,<input alpha1>);
```

Arguments

<output Z>

The output pixel is unsigned, and its width is determined automatically by `gfxBlend` and `gfxBlend2`.

For `gfxBlend` the output, Z, has a width of max(width(X), width(Y)).

For `gfxBlend2` the output, Z, includes the fraction and has a width of (max(width(X), width(Y)) + width(alpha)).

<input X, input Y>

Unsigned input pixels to be blended; can be any width.

<input alpha>

Unsigned blending fraction; can be any width.

<input alpha1>

1-bit unsigned input; the alpha should be all 1s when alpha1 is 1.

Directives

carrysave

Carry-save mode; applicable only for gfxBlend2

fatype

Type of final adder to build (aofcla; cla; clsa; csa; fastcla; ripple; ripple_alt; or auto, the default)

intround

Internal rounding control

round

Rounding control; applicable only for `gfxBlend2`

Description

The `gfxBlend` function mixes, or blends, X and Y to form Z. The alpha and alpha1 determine how much of each input is mixed into the output and are a value between 0 and 1.

The amount of blending, valpha, is computed as follows:

valpha = (alpha + alpha1) / (1 << width(alpha))

For example, to provide an 8-bit resolution, an 8-bit alpha is provided, with the values shown in Table 2-8.

Table 2-8 gfxBlend

alpha1	alpha	valpha
0	00000000	0.0
0	00000001	1/256
...		
0	10000000	128/256 (0.5)
...		
0	11111111	255/256
1	11111111	1.0

After you have valpha, you can specify how X and Y are blended to form Z, by using the following equation:

Z = X * valpha + Y * (1 –valpha)

The `gfxBlend` function performs rounding and truncation automatically; therefore, its output must be in binary and cannot be left in carry-save.

The gfxBlend2 function is similar to gfxBlend. The difference is that gfxBlend2 does not perform rounding and truncation automatically. Therefore, the output, Z, of gfxBlend2 can be left in carry-save format (intround and round can still be applied by the caller).

The output, Z, of gfxBlend2 is full precision, including "width(alpha)" bits of fraction in the LSBs.

To illustrate the relationship between gfxBlend and gfxBlend2, the following example shows how gfxBlend2 can be made identical to gfxBlend:

```
integer aw = 9;
wire [1] a1;
wire [aw] alp; // aw is the width of alp
wire [8] A,B;
directive local (round=aw);
wire [8] Pixout2 = gfxBlend2(A,B,alp,a1) >> aw;
```

The gfxBlend function provides an efficient solution for variable alpha. However, if alpha is a constant, a more direct implementation based on multiplication might be better.

Example

```
wire [1] a1;
wire [9] alp;
wire [8] A,B;
wire Pixout  = gfxBlend (A,B,alp,a1);
wire Pixout2 = gfxBlend2(A,B,alp,a1);
//width of Pixout  is  8
//width of Pixout2 is 17
```

gfxLogicop

Graphics pixel logic processor; performs one of 16 bitwise logical operations on the two inputs

Syntax

```
gfxLogicop(<output Z>, <input I>, <input S>, <input D>);
```

Arguments

<output Z>

Output (width=max(width(S),width(D))

<input I>

4-bit instruction

<input S>

Source input; can be any width

<input D>

Destination input; can be any width

Directive

muxtype

Selects the MUX architecture; should be set to mux

Description

The `gfxLogicop` function performs the 16 possible bitwise functions on the source, S, and the destination, D, to produce the output, Z. The instruction input, I, selects which of the 16 operations is performed, as shown in Table 2-9.

Table 2-9 gfxLogicop Instruction Input, I

I	Z
0	0
1	S&D
2	S&~D
3	S
4	~S&D
5	D
6	S^D
7	S|D
8	~(S|D)
9	~(S^D)
10	~D
11	(S|~D)
12	~S
13	~S|D
14	~(S&D)
15	~0

The implementation of `gfxLogicop` is very efficient. The area and delay correspond approximately to one 4:1 MUX per bit.

---

gfxShift

Graphics five-function shifter/rotator

Syntax

```
gfxShift (<output Z>,<input X>,<input S>,<input MODE>);
```

Arguments

<output Z>

Output is same width as X; always unsigned

<input X>

Input data, width = 2width(s)

<input S>

Shift/rotate amount of input

<input MODE>

3-bit mode control for input

The 3 bits of MODE work independently to specify the operation of `gfxShift`, as shown in Table 2-10 and Table 2-11.

Table 2-10 gfxShift Bit Number

Bit number	Function	Value
0	Right/Left*	1=Right, 0=Left
1	Rotate/Shift*	1=Rotate, 0=Shift
2	Arith/Logical*	1=Arith, 0=Logical

Table 2-11 gfxShift MODE[2]

MODE[2]	Function
x00	Left shift
001	Right logical shift
x10	Left rotate
x11	Right rotate
101	Right arithmetic shift

Note that with left shifting or left rotating, MODE[2] has no effect and is ignored.

Directive

selectop

Type of select (shift input) optimization: msb, lsb, auto

Description

The `gfxShift` function provides the full complement of shift and rotate functions needed in graphics bit manipulation. The input data, X, is shifted or rotated (as determined by MODE) by the amount specified by S.

Note that you can set one or more of the MODE bits to a constant to remove functionality from `gfxShift`. For example, setting MODE[1] to 0 puts `gfxShift` in shift-only mode and saves any logic associated solely with rotating the input.

isolate

Isolates critical nets from heavy loads

Syntax

```
isolate (<output Z>, <input X>);
```

Arguments

<output Z>

Output signal; width/format declared by isolate to be the same as the input

<input X>

Input signal X

Directives

None

Description

The `isolate` function provides an alternative form of the buffering provided by the buffer function. The buffer function builds a buffer tree of the specified depth and then allows access to the buffered signal only.

The `isolate` function inserts a set of noninverting buffers between the input, X, and the output, Z.

Subsequently, access is allowed to the nonbuffered signal (input X) as well as to the buffered signal (output Z), so that more-critical paths can be driven from X and less-critical paths can be driven from Z.

Example

```
input [8] A;
wire [8] ANC; // must match A!
ANC=isolate(A); // ANC has buffer depth 2
buffer (ANC,2); // build buffer tree at output of ANC
```

---

join

Connects signals in a bitwise fashion

Syntax

```
join (<output Z>, <input Di>, ... , <input Dn>);
```

Arguments

<output Z>

Output signal; width/format declared by join to be signed if any inputs are signed and to be as wide as the widest input

<input Di, ..., input Dn>

Input signals to be joined

Directives

None

Description

The `join` function provides a mechanism for electrically connecting wires. Corresponding bits from the inputs are connected.

Typically, the inputs are driven by three-state drivers; if not, a warning is generated.

By default, the output of join is signed if any input is signed.

Example

```
wire [8] A,B;
... // Lines of code here.
wire [8]  C = join(A,B);
// A, B and C are all electrically connected
```

LZ

Counts leading 1s or 0s

Syntax

```
<output Z> = LZ (<input X>[, < integer type >]);

LZ (<output Z>, <input X>[, < integer type >]);
```

Arguments

<output Z>

Output width chosen automatically equals ceil(log2(width(X))) + 1. MSB indicates that X is all 0s or 1s. The remaining bits indicate the number of leading 0s or 1s.

<input X>

Input; width and format determined by the caller

<integer type>

Indicates value of the bits to count: 0 for leading 0s (default), 1 for leading 1s

Directives

None

Description

The `LZ` function counts leading 0s or leading 1s from the input, X, to form the output, Z. Counting starts from the MSB of X. This is effectively a priority encoder.

The LSB of Z indicates the number of leading 0s or 1s, and the MSB indicates whether X is all 0s or all 1s. Note that when the MSB of Z is set, the LSBs of Z are set to all 1s for leading-0 detection but are set to all 0s for leading-1 detection.

The `LZ` function uses a very fast architecture compared to `norm`. Its behavior is similar to `norm`, but it does not normalize the input as `norm` does.

If the width of X is not a power of 2, X is left-shifted to make its width a power of 2. Because the padding for left shift is 0, the behaviors of leading-0 and leading-1 detection are not symmetrical.

Examples

The following two examples show the operation of the LZ function. The width of the output, Z, is ceil(log2(width(X))) + 1. The example results are shown in Table 2-12 and Table 2-13.

```
wire [6] X;
 . . .
Z=LZ(X); //detect leading zeros
```

Table 2-12 First LZ Example Results

Input X	Left shift	Output Z
001100	00110000	0010
000000	00000000	1111
00000000	00000000	1111

```
wire [6] X;
 . . .
Z=LZ(X,1);
```

Table 2-13 Second LZ Example Results

Input X	Left shift	Output Z
110011	11001100	0010
111111	11111100	0110
11111111	11111111	1000

Note that for the X = 111111 case, the MSB of Z is not set, because it is not all 1s after the left shift.

---

mac, maccs

mac

Multiplier-accumulator signal function

maccs

Multiplier-accumulator signal function with carry-save

Syntax

```
mac     (<output Z>, <input X>, <input Y>, <input R>,
        <input S>);

maccs   (<output Z>, <input X>, <input Y>, <input R>,
        <input S>);
```

Arguments

<output Z>

Output signal; width/format declared by the caller

<input X>

Input signal named X; value to accumulate

<input Y>

Input signal named Y; value to accumulate

<input R>

Input signal named R; 1-bit unsigned, active-low reset

<input S>

Input signal named S; multiplier-accumulator reset value

Directives

carrysave

Must be set to off

fadelay

Final adder delay goal, in ps

fatype

Type of final adder to build (aofcla; cla; clsa; csa; fastcla; ripple; ripple_alt; or auto, the default)

maxtreedepth

Maximum Wallace tree depth

multtype

Multiplier type

pipeline

Enables/disables automatic pipelining

round

Must be set to 0 (use the S input of mac or maccs to implement rounding)

Description

The `mac` and `maccs` functions generate multiplier-accumulators. The width of the accumulator is controlled by the width of the output, Z.

The X and Y inputs can have independent widths and formats. A reset input, R, is provided for resetting the accumulator to the start value, S, plus the product of the inputs.

The `mac` function provides a smaller implementation by using a carry-propagate adder for the accumulator. The `maccs` function uses a carry-save adder for the accumulator, allowing higher-speed operation with greater area. Both structures can be pipelined.

The exact function implemented by `mac` and `maccs`, ignoring latency generated by automatic pipelining, is given by

$$Z(t) = X(t-1) * Y(t-1) + (R(t-1) \; ? \; Z(t-1) : S(t-1))$$

Example

```
input [16] X;
input signed [14] Y;
input [1] R;
Z=mac(X,Y,R,0); // simple MAC, reset to 0
```

mag

Computes the absolute value of a signal

Syntax

```
mag (<output Z>, <input X>);
```

Arguments

<output Z>

Output signal; width/format declared by the caller

<input X>

Input signal X

Directives

fatype

Type of final adder to build (aofcla; cla; clsa; csa; fastcla; ripple; ripple_alt; or auto, the default)

fadelay

Final adder delay goal, in ps

maxtreedepth

Maximum Wallace tree depth

direxT

Forces direct sign extension

carrysave

Carry-save mode

round

Rounds result to given position

Description

The `mag` function is the signal equivalent of the `abs` function. (For information about the `abs` function, see “abs” on page 2-84). The `mag` function is based on the `sgnmult` function and computes the absolute value of the input: The output, Z, equals the input, X, if the input is greater than or equal to 0. Otherwise, Z equals -X.

The width of Z must be declared by the caller. However, `mag` can be embedded in another arithmetic expression if it is being added or subtracted with another signal.

The resulting hardware is small and fast, because `mag` does not use any carry-propagate adders. Other operations, such as multiplication, require the generation of a temporary variable for the output, Z. This temporary variable cannot be automatically generated, because its size needs to be supplied by the designer.

Example

```
input signed [16] X,Y;
wire [15] Z0;
wire signed [17] Z1;
Z0 = mag(X); //Z = -X if X < 0, Z = X if X >= 0
Z1 = mag(Y) + X;
```

max2, maxmin, min2

max2

Maximum value signal function

maxmin

Maximum and minimum value signal function

min2

Minimum value signal function

Syntax

```
max2  (<output Max>, <output XGEY>, <input X>, <input Y>);

min2  (<output Min>, <output XGEY>, <input X>, <input Y>);

maxmin  (<output Max>, <output Min>, <output XGEY>,
         <input X>,<input Y>);
```

Arguments

<output Max>

Output signal that holds the maximum value; width/format declared by the function

<output Min>

Output signal that holds the minimum value; width/format declared by the function

<output XGEY>

Output signal that indicates if X >= Y; width/format declared by the function

<input X>

Input signal

<input Y>

Input signal

Directives

direxT

Forces direct sign extension

round

Rounds result to given position

Description

The `max2` function sets the signal output (max) to the maximum of the two inputs (X, Y) and sets the output XGEY to 1 if X is greater than or equal to Y.

The `min2` function sets the output (min) to the minimum of the two inputs (X, Y) and sets the output XGEY to 1 if X is greater than or equal to Y.

The `maxmin` function provides both maximum (max) and minimum (min) values.

The width of the output(s) is the width of the largest input. If the largest input is unsigned or the inputs are of equal size and another input is signed, the width of the output is the width of the largest (or equal sized) input + 1. The format of the output(s) is signed if any of the inputs are signed; otherwise, the format is unsigned. XGEY is always a 1-bit unsigned signal.

Example

```
max2(z, XGEY, x, y);
//z= max(x,y), XGEY=(x>=y)

min2(z, XGEY, x, y);
//z= min(x,y), XGEY=(x>=Y)

maxmin(m, n, XGEY, x, y);
//m=max(x,y), n=min(x,y), XGEY=(x>=y)
```

---

multp

Multiplies the plus constant: computes (X * (Y + 1-bit signal))

Syntax

```
multp (<output Z>, <input X>, <input Y>, <input Q>);
```

Arguments

<output Z>

Output signal; width/format declared by the caller

<input X>

Input signal X

<inputY>

Input signal Y

<input Q>

Input signal Q; must be 1-bit unsigned

Directives

carrysave

Carry-save mode

dirext

Forces direct sign extension

fadelay

Final adder delay goal, in ps

fatype

Type of final adder to build (aofcla; cla; clsa; csa; fastcla; ripple; ripple_alt; or auto, the default)

intround

Internal rounding control

maxtreedepth

Maximum Wallace tree depth

round

Rounds result to given position

Description

The `multp` function multiplies a signal by another signal summed with a 1-bit signal. It provides a more efficient implementation than the direct description: Z = X * (Y + Q).

The width of output Z must be declared by the caller. However, `multp` can be embedded in another arithmetic expression if it is being added to or subtracted from another signal.

Other operations, such as multiplication, require the generation of a temporary variable for the output, Z. This temporary variable cannot be automatically generated, because its size is supplied by the designer.

The `multp` function uses Booth architecture but might not provide the benefits of this architecture if the Booth cells are poor or the multiplier is small.

Example

```
input X, Y, Q;
//function inputs; Q is 1-bit unsigned

wire signed [1] S; //product sign

Z1 = multp(X, Y, Q); //Z1=X*(Y+Q)

Z3 = multp(X, ~Y, 1);
//Z3 = -X*Y (efficient way to compute negative product)

Z4 = multp(X, Y^S, S);
//Z4 = S ? -X*Y:X*Y is an
//efficient way to compute positive/negative product)

directive (carrysave = "on");

Z2 = multp(X, Y, Q); //Z2 is a carrysave
```

---

norm, norm1

norm

Converts a signal to floating-point format, using leading 0s

norm1

Converts a signal to floating-point format, using leading 1s

Syntax

```
norm (<output M>, <output E>, <input X>);

norm1 (<output M>, <output E>, <input X>;
```

Arguments

<output M>

Output signal that stores the mantissa; width/format declared by the caller

<output E>

Output signal that stores the exponent; width/format declared by the caller; format must be unsigned

<input X>

Input signal X

Directives

None

Description

The `norm` and `norm1` functions convert an unsigned integer to floating-point format. These functions detect the number of leading 0s or 1s in the input, X, and shift the input left by this amount.

The number of leading 0s or 1s is the exponent (output E) of the normalized number, and the shifted number is the mantissa (output M = X << E). The `norm` function detects leading 0s, whereas `norm1` detects leading 1s.

If the width of X is not a power of 2, X is left-shifted to make its width a power of 2. The shifted input is then normalized. Finally, M is shifted right by the same amount as the left shift. These operations affect the results when the input is all 0s (`norm`) or all 1s (`norm1`).

The exponent cannot be declared as signed. The widths of the exponent and the mantissa are declared by the caller of `norm` or norm1. The width of the exponent controls the maximum number of shifts used to normalize the input. Both functions issue a warning if the exponent width is greater than log2(width(X)).

The `norm` and `norm1` functions cannot be embedded in other expressions, because embedding requires the generation of a temporary signal for the mantissa (output M). This temporary signal cannot be automatically generated, because its size needs to be supplied by the designer.

Table 2-14 shows the operation of `norm`—leading 0s—for exponent operands 2 bits wide and 3 bits wide. Note that the exponent in the last row is 7, not 5.

Table 2-14 Norm

Input	Output (mantissa)	Output (exp)
10000000	10000000	0
01110101	11101010	1
00001110	01110000	3
		. . .
00000	00000	7

Table 2-15 shows the operation of `norm1`—leading 1s—for exponent operands 2 bits wide and 3 bits wide. Note that the exponent in the last row is 5, not 7.

Table 2-15 Norm1

Input	Output (mantissa)	Output (exp)
01111111	01111111	0
10001010	00010100	1
11110001	10001000	3
		. . .
11111	00000	5

Note that in some of the cases, the output is not fully normalized, because the shift output is not wide enough.

Example

```
input IN;          // function input
wire [32] MANT; // width must be declared here
wire [4] EXP;   // must be unsigned
norm (MANT, EXP, IN);
```

registers: eqreg, eqreg1, eqreg2, ensreg, preg, sreg

eqreg

Latency equalization register function; latency = maximum of reference list

eqreg1

Latency equalization register function; latency = as given

eqreg2

Latency equalization register function; latency = sum of reference list

ensreg

State register with enable

preg

Pipeline register function

sreg

State register function

Syntax

```
eqreg  (<output Z>, <input X>, <integer n>, <input X1>, ...,
       <input Xn>);
eqreg1 (<output Z>, <input X>, <integer m>);
eqreg2 (<output Z>, <input X>, <integer n>, <input X1>, ...,
       <input Xn>);
```

Arguments

<output Z>

Output signal; width/format determined from the width and format of the input signal X

<input X>

Input signal named X

<integer n>

Number of signals in the reference list

<integer m>

Desired latency

<input X1>

Input signal representing the first reference signal

<input Xn>

Input signal representing the last reference signal

Syntax

```
ensreg  (<output Z>, <input X>, <input en>, <integer n>,
         <output Y0>, ..., <output Yn>);

preg    (<output Z>, <input X>, <integer n>, <output Y0>,
        ..., <output Yn>);

sreg    (<output Z>, <input X>, <integer n>, <output Y0>,
        ..., <output Yn>);
```

Arguments

<output Z>

Output signal; width/format determined from the width and format of the input signal X

<input X>

Input signal named X

<input en>

Input signal representing enable

<integer n>

Length of the register; default value = 1

<output Y0>

Output signal representing 0th tap; same as X

<output Yn>

Output signal representing *n*th tap; same as Z

Directive

delstate

Specifies the number of state registers to loan for pipelines

Description

Several functions are available for generating registers. The equalization registers (`eqreg`, `eqreg1`, and `eqreg2`) are pipeline registers, which equalize (match) the latency of a signal to a reference value. Module Compiler automatically sets the length of these registers to satisfy the latency goal.

The `eqreg1` function specifies the latency goal as an absolute quantity. The latency goal in `eqreg` is the maximum of the latencies of the signals in the reference list (X1, ... Xn). The latency goal in `eqreg2` is the sum of the latencies of the signals in the reference list.

The state registers (`ensreg` and `sreg`) create delays as specified by their arguments in order to meet the requirements of the algorithm being implemented. These registers do not add any latency to the circuit. The `sreg` function creates a straightforward register of length *n*.

The output, Z, is a copy of the input, X, delayed by *n* clock cycles. The `ensreg` function performs that same function, except that it also accepts an enable input *en* that is a 1-bit unsigned signal. If enable

is high, the data input is shifted in the same manner as `sreg`; if enable is low, no shift takes place and the output is held at the present value.

Both `ensreg` and `sreg` provide access to all the shift register outputs, Y0 through Yn. Y0 is connected to the input, X; Y1 to the input delayed by one cycle; Y2 to the input delayed by two cycles; and so on. In all cases, the called function creates outputs of the same width and format as the input.

A pipeline register of length *n* increases the latency by *n*. The usage of `preg` is identical to that of `sreg`.

Example

```
Z=eqreg(A,3,B,C,D);
// length is equal to max(lat(B),lat(C),lat(D))-lat(A)

Z=eqreg1(A,3); // length is equal to 3-lat(A)

Z=eqreg2(A,3,B,C,D);
// length is equal to lat(B)+lat(C)+lat(D)-lat(A)

Z=sreg(A,1); // 1 tap state register

Z=sreg(A,4,Y_0,Y_1,Y_2,Y_3,Y_4);
//have access to all taps, Y_0 is same as input A, Y_4 is
final output and same as Z

Z=ensreg(A,Y[3],4); // 4-tap enable state reg, enabled on
Y[3]

input [8] A,B;
input [16] C;
wire [16] prod = A * B;
wire [16] temp = preg (prod, 1);
//temp is prod delayed 1
//preg creates latency 1
wire [16] Z = temp + C;
//Module Compiler inserts pipelines at C,
//so both inputs have the same latency
```

ResolveLatency, ResolveLatencyLoop

Pipelines the design to a user-defined latency level

Syntax

```
ResolveLatency (<output Q>, <input D>, <latency>);

ResolveLatencyLoop (<output Q>, <input D>, <latency>);
```

Arguments

<output Q>

Output signal Q; width/format are inherited from D (declared by the function)

<input D>

Input signal D; width/format declared by the caller

<latency>

User-defined latency; no default, must be specified

For ResolveLatency: latency > = 0

For ResolveLatencyLoop: latency > = 1

Directives

None

Description

The `ResolveLatency` function is used at the end of a pipelined section of code, whereas the `ResolveLatencyLoop` function is used at the end of a pipelined section of code when the output is a feedback point, such as an interleaved accumulator. For both of these functions, the desired latency must be specified.

Use `ResolveLatency` for a feed-forward path of a design. The latency specified can be 0 or greater. `ResolveLatency` does not guarantee that the output is registered.

Use the `ResolveLatencyLoop` function at the end of a feedback loop. The latency specified can be 1 or greater. `ResolveLatencyLoop` ensures that the output is registered. You can also use `ResolveLatencyLoop` at the end of a feed-forward path to ensure that the output is registered.

For both functions, the output, Q, is forced to the specified latency and the Module Compiler latency counter is reset to 0. This prevents any latency deskewing outside this section of code, and it forces Q to be cycle accurate in the RTL model. For additional information about `ResolveLatency` and `ResolveLatencyLoop`, see "Handling Latency" in Chapter 10 in the *Module Compiler User Guide*.

Example

```
// module example
(Z,X,Y,Reset_n,w,ovf,fa,latm,lata,pslm,psla);
integer w = 8; // input width
integer ovf = 2; // extra accumulator bits
integer accw = 2*w + ovf; // accumulator width
integer latm = 0; // multiplier pipeline latency
integer lata = 2; // adder pipeline latency
integer pslm=0,psla=0; // pipe slack
string fa   = "csa"; // final adder type
input signed [1] Reset_n; // accumulator reset
input [w] X,Y;
output [accw] Z;
directive(fatype=fa,pipeline="on");

directive(pipeslack=pslm);
wire [2*w] prod = X*Y;
wire prodRL = ResolveLatency(prod,latm);

directive(pipeslack=psla);
wire [accw] ACCin = prodRL + Z&Reset_n;
Z = ResolveLatencyLoop(ACCin,lata);

endmodule
```

sat, sati

sat

Signal clipping function

sati

Signal clipping function with inverted output

Syntax

```
sat (<output Z>, <input X>);

sati (<output Z>, <input X>);
```

Arguments

<output Z>

Output signal; width/format declared by the caller

<input X>

Input signal X

Directives

None

Description

The `sat` and `sati` functions implement a signal clipping function. If the value at the input, X, exceeds the maximum value representable at the output, Z, the output takes on the value of the maximum value.

If the value at the input is less than the minimum value representable at the output, the output takes on the minimum value. If the input value is between the minimum and the maximum, the output is a copy of the input. The `sati` function inverts the final result, whereas `sat` returns the true result.

Both functions require the caller to specify the size and the format of the output, Z. This size determines the minimum and maximum possible values. Consequently, `sat` and `sati` cannot be embedded in other expressions, because this embedding requires the generation of a temporary variable for the output, Z. This temporary variable cannot be automatically generated, because its size needs to be supplied by the designer.

Example

```
input signed [8] X; // module input with preassigned value
wire [4] Z1, Z2, Z4;
wire [8] Z4;
Z1=sat(X); / clip input values to {0 to 15} range
Z2=sati(X); // same as Z2 = ~sat(X);
Z3=sat(X);
Z4=sat(X[7:4]);
```

SetLatency

Specifies the latency of a module input signal.

Syntax

```
SetLatency (<output Z>, <input A>, <latency>);
```

Arguments

<output Z>

Output signal Z; width/format are inherited from A

<input A>

Input signal A; width/format declared by the caller

<latency>

User-defined latency; (>=0)

Directives

None

Description

Use the `SetLatency` function to specify input signal latencies. This function works only on input signals and only increments the internal latency counter; it does not create any registers. Accordingly, only the Module Compiler latency counter is incremented.

Module Compiler automatically performs latency deskewing by inserting pipeline registers to balance the signal latencies when signals merge together in the design.

Example

```
input [8] a, b, c;
/* assume a has latency=5, b has latency=2, c has latency=0 */

SetLatency (a, 5);
/* now input a to five levels of latency, no FF insertion */

SetLatency (b, 2);
/* set input b to two levels of latency, no FF insertion */

/* setting signal c with latency=0 is optional */

SetLatency (c,0);
```

sgnmult

Sign-multiplier: returns X if Y (select) is 0 and -X if Y (select) is 1.

Syntax

```
sgnmult ( <output Z>, <input X>, <input Y> );
```

Arguments

<output Z>

Output signal; width/format declared by the caller

<input X>

Input signal X

<input Y>

Input signal Y; must be 1-bit

Directives

carrysave

Carry-save mode

dirext

Forces direct sign extension

fadelay

Final adder delay goal, in ps

fatype

Type of final adder to build (aofcla; cla; clsa; csa; fastcla; ripple; ripple_alt; or auto, the default)

maxtreedepth

Maximum Wallace tree depth

round

Rounds result to given position

Description

The `sgnmult` function multiplies a signal by plus or minus 1, based on a control signal, and generates –X if Y is 1 and X if Y is 0.

The width of the output, Z, should be declared by the caller. However, `sgnmult` can be embedded in another arithmetic expression if it is being added to or subtracted from another signal.

Other operations, such as multiplication, require the generation of a temporary variable for the output, Z. This temporary variable cannot be automatically generated, because its size needs to be supplied by the designer.

The input, X, can be in binary or carry-save form, as shown in the following example. The output, Z, can also be in binary or carry-save form, depending on the carry-save directive.

```
input X, S; // function inputs; S is 1-bit unsigned
Z1=sgnmult(X, S); // Z1=+X if S=0, Z1=-X if S=1
directive (carrysave = "on");
Z2 = sgnmult(X, S); // Z2 is a carrysave
```

The following example shows `sgnmult` in the carry-save context. The example models z = a * b +/-(c +/-(a + b)) + c + d, with the inner addition and subtraction controlled by the signed 1-bit signal s0 (where s0 = a[0] > b[0]) and the outer addition and subtraction controlled by the signed 1-bit signal s.

```
module mod(z,a,b,c,d,s);
input signed [8] a,b,c,d;
input signed [1] s;
output [19] z;
directive(fatype="aofcla",multtype="booth");
directive(carrysave="on");
wire signed [1] s0 = a[0]>b[0];
wire  signed [17] tmp1 = a*b + sgnmult(c + sgnmult(a+b, s0), s);
wire signed [18] tmp2 = tmp1 + c;
directive(carrysave="off");
z = tmp2 + d;
endmodule
```

shiftlr

Shift left/right function

Syntax

```
shiftlr(<output Z>, <input X>, <input SHIFT>, <input LEFT>,
   <input LOG>);
```

Arguments

<output Z>

Output signal; format is set by `shiftlr` to match X

<input X>

Data input signal

<input SHIFT>

Shifts input signal

<input LEFT>

Single-bit direction input; shifts left when TRUE

<input LOG>

Single-bit mode input; shifts logical when TRUE

Directives

None

Description

The `shiftlr` function is used when an input must be shifted both left and right.

Two control signals (LEFT and LOG) determine the runtime operation the function performs.

The active-high input LEFT controls the direction, and the active-high input LOG controls the right-shift mode.

Right shifts can be either arithmetic (sign extended) or logical (zero extended). Left shifts are always zero extended.

The modes are summarized in Table 2-16.

Table 2-16 shiftlr

LOG	LEFT	Function
X	1	Left shift
0	0	Right arithmetic shift
1	0	Right logical shift

Example

```
wire [32] Z,X;
wire [6] S;
wire [2] CNT;
// shift X S-places. direction = left if CNT[1]==1,
// right if CNT[1]==0
// if right shift, shift logical if CNT[0]==1,
// arithmetic if CNT[0]==0
Z=shiftlr(X,S,CNT[1],CNT[0]);
```

Supporting Functions

The following functions define datapaths and perform conditional synthesis on them:

- abs
- critpath, critmode, enablepath, disablepath
- csconvert
- directive
- error, fatal, warning, info
- fnArgs
- hidelat
- formatStr
- isEven, isOdd
- is2expN, is2expNmin1
- max, min
- param
- showgroup
- string, string constant
- width, log2

abs

Absolute-value integer function

Syntax

```
<integer> = abs(<integer expression>);
```

Directives

None

Description

The `abs` function returns the absolute value of the given integer. The absolute value of an integer is the value without regard to its sign.

For information about the equivalent function for signal variables, see “mag” on page 2-59.

Example

```
integer x = -10;
integer y = abs(x); // y equals 10
```

critpath, critmode, enablepath, disablepath

critpath

Analyzes a path

critmode

Sets critical path reporting mode

enablepath

Enables paths through a given point

disablepath

Disables paths through a given point

Syntax

```
critpath ( <string start>, <string end>, <string name> );
```

Arguments

<string start>

String specifying the name of the startpoint

<string end>

String specifying the name of the endpoint

<string name>

String specifying the name for reporting purposes

Syntax

```
critmode (<string mode >);
```

Arguments

<string mode>

String specifying the reporting mode, either full or short

Syntax

```
enablepath (<string point>);

disablepath (<string point>);
```

Arguments

<string point>

String specifying the points that should be enabled/disabled

Directives

None

Description

The functions `critpath`, `critmode`, `enablepath`, and `disablepath` are positionally dependent and are executed in the order listed in the input description.

The `critpath` function specifies a path to be analyzed. The path goes from start to end and does not go through disabled intermediate points.

The `critmode` function sets the current reporting mode. Startpoints and endpoints can be the name of an operand, a bit range, or *. Use * to match any module input or any output. In addition, you can use ** as the endpoint to match any endpoint, such as a module output or an internal endpoint at a flip-flop.

The `critmode` function accepts either full or short as the mode. A full critical path is printed if the current mode is full. Only the delay at the path endpoint is printed if the mode is short. The mode can be changed as often as desired. The default mode is full.

The `disablepath` function prevents paths from passing through the supplied point. The `enablepath` function reenables any points that were disabled with `disablepath`. Both accept an operand name: bit range or *, which indicates all internal paths. By default, no points are disabled.

Example

```
module foo (A,B,D,F);
input [8] B;
directive (indelay=3000,logopt="off");
input [8] A;
wire [8] A1=isolate(A);
wire [8] A2=isolate(A1);
wire [8] A3=isolate(A2);
wire [8] A4=isolate(A3);
output [8] D=A4;
wire [8] C=A+B;
wire [8] E=isolate(C);
output [8] F=E;

critpath("A","*","A_to_anywhere");
disablepath("F");
critpath("A","*","A_to_anywhere_but_F");
disablepath("D");
critpath("A","*","A_to_anywhere_but_F_or_D");
enablepath("D");
enablepath("F");
disablepath("E");
critpath("A","*","A_to_anywhere_but_E");
enablepath("E");
disablepath("C");
critpath("A","*","A_to_anywhere_but_C");
enablepath("C[5]");
critpath("A","*","A_to_anywhere_but_C[5:7]");
enablepath("C[7:5]");
critpath("A","*","A_to_anywhere");

critmode ("short");
critpath ("A2", "A3", "path2");
critpath ("A2[3]", "A3[3]", "path3");
critpath ("A2[3]", "A3[2]", "path4");
critpath ("*", "A3", "path5");
critpath ("B", "A3", "path6");
critpath ("A4", "A1", "path7");
critpath ("A1", "A4", "path8");
critpath ("A1", "*", "path9");
critmode ("full");
critpath ("A2", "A3", "path2");
critpath ("A2[3]", "A3[3]", "path3");
critpath ("A2[3]", "A3[2]", "path4");
```

```
critpath ("*", "A3", "path5");
critpath ("B", "A3", "path6");
critpath ("A4", "A1", "path7");
critpath ("A1", "A4", "path8");
critpath ("A1", "*", "path9");
endmodule
```

---

csconvert

Accesses the sum and carry terms of a carrysave operand.

Syntax

```
<csconvert> = (<output S>, <output C>, <input A>);
```

Argument

<output S>

Sum term of a carrysave operand; same format as A; width must be determined from the design report after the initial run.

<output C>

Carry term of a carrysave operand; same format at A; width must be determined from the design report after the initial run.

<input A>

Input operand in carry-save format.

Directive

carrysave

Must have been set to convert during generation of the input carrysave operand; however, it can be set to any value during the call to `csconvert`.

Description

The `csconvert` function accesses the sum and carry terms of a carrysave operand. Full instructions governing its usage are described in the *Module Compiler User Guide*.

This function is only necessary when carry save signals must be gated, or MUXed, in a design. You cannot gate, or MUX, a normal carrysave signal. Instead, first access the sum and carry terms from the carrysave operand that holds the result of the arithmetic expression. These sum and carry terms are gated, or MUXed, in parallel. You obtain the final binary result by adding together the sum and carry terms with carrysave mode set to off.

The carrysave operand, used as the input to `csconvert`, must be generated with the carrysave mode set to convert; however, `csconvert` can be called in any carrysave mode (including off) to obtain the sum and carry terms.

Note:

Using `csconvert` is recommended only for experienced datapath designers. Synopsys recommends that you have experience with datapath design in Module Compiler prior to using this function. If you do not use this function carefully, an RTL-versus-gate mismatch can occur.

Example

```
module acc(Z,X,RESET);
  input signed [8] X;
  output signed [8] Z;
  input [1] RESET;
  wire signed [8] ACC0,ACC1,X1,XPR,ZA,RZA0,RZA1;
  wire signed [10] ZA0,ZA1; //determined from acc.report
  ACC0=sreg(RZA0);//need two sreg calls for carrysave
  ACC1=sreg(RZA1);

  directive(carrysave="convert");
  //must use the convert option here
  ZA=X+ACC0+ACC1;
  csconvert(ZA0, ZA1, ZA);
  //generate two signed signals,ZA0,ZA1

  directive(MUXtype="andor");
  //now we can MUX the carrysave signal
  RZA0=RESET ? ZA0 : 0; //to allow the loop to be reset
  RZA1=RESET ? ZA1 : 0;

  directive(carrysave="off",fatype="clsa");
  Z = ACC0 + ACC1;
endmodule
```

---

directive

Queries the value of a synthesis attribute

Syntax

```
<string> = directive(<attribute>);
```

Argument

<attribute>

Synthesis attribute normally set with other form of directive

Directives

None

Description

The `directive` function, when used in this form, is used to query the value of a synthesis attribute. It returns a string value representing the value assigned to the synthesis attribute.

Note that `directive` can be used in one of two ways. In the form described here, it returns the value of a single attribute. In its other form, it is used to set the value of one or more synthesis attributes. For more information about the other form of `directive`, see Chapter 3, “Directives”.

Example

```
//use directive to save the current context
string oldcs = directive(carrysave);
string oldpipe = directive(pipeline);

//set the desired attributes for this section
directive (pipeline="off", carrysave="convert");
.
. <perform arithmetic here>
.
//now restore the original context
directive (carysave=oldcs, pipeline=oldpipe);
```

error, fatal, warning, info

error

Prints an error message

fatal

Prints a fatal error message

warning

Prints a warning message

info

Prints an information message

Syntax

```
<keyword> (<arg list>);
```

Arguments

<keyword>

error | info | warning | fatal

<arg list>

<expression> [, <expression>]*

Directives

None

Description

These built-in functions perform debugging and error checking.

The printing behavior of these functions is identical. If they encounter a nonsignal variable or expression, they print its value. If they encounter a signal name, they print its hierarchical name.

The processing behavior of these functions is different. The `info` function has no effect on subsequent processing. The `warning` function allows processing to continue but increments the number of warnings reported at the end.

The `error` function increments the number of parse errors; parsing continues, but no synthesis takes place. If there are too many errors, the parser exits as well. The `fatal` function suspends parsing immediately.

Example

```
function someFunction (..., X, ...)
...
info ("X is: ", X);
// prints: "X is A & width is 8" if someFunction was
// called with signal A of width 8

info (" & width is: ", width(X), "\n");

error("x + y is less than zero :", x +y, "\n");
// prints "x + y is less than zero: <value>"
...
endfunction
```

---

fnArgs

Returns the number of arguments passed to a function

Syntax

```
fnArgs( )
```

Directives

None

Description

The `fnArgs` function returns an integer representing the number of arguments passed to a function. It is commonly used in functions with a variable number of arguments.

Example

```
function sum (Z, repl(i,fnArgs()-1,","){X{i}});
input repl(i,fnArgs()-1,","){ X{i}};
output [8] Z=repl(i,fnArgs()-1,"+"){X{i}};
endfunction

module adder (Z,A,B,C);
input [8] A,B,C;
output [8] Z;
Z=sum (A,B,C);
endmodule
```

---

hidelat

Hides the latency of a signal

Syntax

```
hidelat (<output Z>, <input X>);
```

Arguments

<output Z>

Output signal; width/format declared by `hidelat` to be the same as the input

<input X>

Input signal X representing data

Directives

None

Description

Module Compiler can automatically deskew any latency differences created by pipelining.

The `hidelat` function is used to hide the latency of a signal to prevent automatic deskewing.

The output (Z) is created by `hidelat` with the same size and format as the input. The output latency is set to `0`.

Example

```
input signed [7:0] X;                //module input with preassigned value
wire signed [7:0] ACC,X1,XPR,ZA,RZA;
XPR=preg (X,2);                      //XPR has a latency of 2
X1=hidelat (XPR);                  //need to hide latency of XPR before the loop
ACC=sreg (ZA, 1);
ZA=X1+ACC;                          //latency of X1 is zero, same as ACC
```

---

formatStr

Returns the format of a signal as a string

Syntax

`formatStr (< signal >)`

Directives

None

Description

The `formatStr` function returns the format as a signed or unsigned string.

Example

```
string x = formatStr(A);
wire {x} [8] B;// B has same format as A
```

---

isEven, isOdd

isEven

Returns 1 if the input integer is even

isOdd

Returns 1 if the input integer is odd

Syntax

```
isEven (< integer expression >)
```

```
isOdd (< integer expression >)
```

Arguments

<integer expression>

Value must be a nonzero positive integer

Directives

None

Description

The `isEven` function returns 1 if the input integer is even. The `isOdd` function returns 1 if the input integer is odd. For both the `isEven` and `isOdd` functions, the input integer must be a nonzero positive integer.

The following example demonstrates the use of the `isEven` and `isOdd` functions.

Example

```
function SYMMFIR (Z, X, repl(i,fnArgs()-2,","){ C[i] });
integer taps = fnArgs()-2;
if (isOdd(taps))
     {
        . //MCL description of a symmetric FIR
          //with odd # of taps
        .
           }
else
      {
        . //MCL description of a symmetric FIR with
          //even # of taps
        .
           }
endfunction
```

In the previous example, the SYMMFIR function selects between different symmetric FIR architectures. Depending on whether the number of taps is even or odd, an optimum FIR architecture can be selected.

When a symmetric FIR has an even number of taps, the multipliers can be shared. When a symmetric FIR has an odd number of taps, the multiplier associated with the central tap cannot be shared; therefore, a separate architecture can be defined when the number of taps is odd.

---

is2expN, is2expNmin1

is2expN

Returns 1 if the input integer is a power of 2

is2expNmin1

Returns 1 if the input integer is 1 less than a power of 2

Syntax

```
is2expN   (< integer expression >);

is2expNmin1 (< integer expression >);
```

Arguments

<integer expression>

Value must be a nonzero positive integer

Directives

None

Description

The `is2expN` function determines whether a given integer is a power of 2. If the integer satisfies this criterion, the `is2expN` function returns 1.

Likewise, the `is2expNmin1` function determines whether a given integer is 1 less than a power of 2. If the integer satisfies this criterion, the `is2expN` function returns 1.

For both the `is2expN` and the `is2expNmin1` functions, the input integer must be a nonzero positive integer. The following example shows the use of the `is2expN` function:

Example

```
if (!is2expN(width(A)))
 {
  warning ("Width of A is not a power of 2");
     }
```

The previous example uses the `is2expN` function to ensure that all width declarations made in your Module Compiler Language code are a power of 2. If your code uses the bit range declaration style to specify signal widths (for example, wire [m:0] A), use the `is2expNmin1` function instead for testing purposes.

---

max, min

max

Maximum-value integer function

min

Minimum-value integer function

Syntax

```
<integer> = max (<integer expression>,
                 <integer expression>);

<integer> = min (<integer expression>,
                 <integer expression>);
```

Directives

None

Description

The `max` function returns the greater of the two integer value arguments. The `min` function returns the lesser of the two integer arguments.

For information about the equivalent function for signal variables, see “max2, maxmin, min2” on page 2-61.

The width of the output(s) is the width of the largest input. If the output and the largest input have different formats, the width of the output is the width of the largest input + 1. XGRY is always a 1-bit unsigned signal.

---

param

Parameter iteration function

Syntax

```
param(par0,val0,par1,val1, ...);
```

Arguments

<par1>

String with the name of parameter 1

<val1>

Integer or string value of parameter 1

Directives

None

Description

In the simple case, an individual set of parameter values for the module being built is specified in the Parameters input field of the GUI. When many different sets of parameter values are of interest, parameter iteration can be used.

The `param` function is used within a parameter iteration file to specify parameter name and value pairs. The name of the parameter iteration file is specified in the Par Iter File input field of the GUI.

The following example shows a sample parameter iteration file that generates 12 sets of parameter values for a module with three parameters. The module has parameters named mod, arch, and w. The values of arch and w are used to build the value of mod.

Twelve distinct permutations of arch and w are generated with the loops in the parameter iteration file. The parameter iteration file exists outside the module being built.

Example

```
module iter();
integer arch,w;
string name; //permute arch={0,1,2} with w={8,16,24,32}
repl(arch,3) {
    replicate(w=8; w<=32; w=w+8) {
        name = string("RINK_", arch, "_" ,w);
        param("mod", name, "arch", arch, "w", w);
    }
}
endmodule
```

---

showgroup

Displays group information

Syntax

```
showgroup(<string name>);
```

Argument

<string name>

The name of the group to print

Directives

None

Description

The `showgroup` function displays hierarchical group information. It accepts a string argument with a group name pattern of the form

```
<name>[.<name>[.<name>] ... ]
```

where <name> can be a literal name or *.

The * matches all names at that level of hierarchy. All groups that match the pattern are printed in the report file. If more than one group matches the pattern, the statistics for the entire set of groups are also printed.

Example

```
module video (taps,replicate(integer i=0; i<taps; i=i+1)
{YC{i},}R,G,B,Y,U,V);
integer taps;
directive (pipeline="on",delay=9999999);
input signed [8] replicate(i=0; i<taps; i=i+1) {YC{i},};
input [8] R,G,B;
output [21] Y,U,V;
buffer(R,2);
buffer(G,2);
buffer(B,2);
wire signed [16] U1,U_int,V1,V_int;
wire [16] Y1,Y_int;
directive (group="matrix.Y");
Y_int=R*89+G*138+B*47;
directive (group="matrix.U");
U_int=0-33*R+144*G+88*B;
directive (group="matrix.V");
V_int=53*R-91*G+102*B;
directive (group="hide.Y");
Y1=hidelat(Y_int);
directive (group="hide.U");
U1=hidelat(U_int);
directive (group="hide.V");
V1=hidelat(V_int);
wire unsigned [10] YSR, replicate(i=0; i<=taps; i=i+1)
{Y_{i},};
```

```
wire signed [10] USR,replicate(i=0; i<=taps; i=i+1)
{U_{i},};
wire signed [10] VSR,replicate(i=0; i<=taps; i=i+1)
{V_{i},};
directive (group="fir.Y");
YSR=sregtaps(Y1[15:6],taps,replicate(i=0; i<=taps; i=i+1)
{Y_{i},});
directive (group="fir.U");
USR=sregtaps(U1[15:6],taps,replicate(i=0; i<=taps; i=i+1)
{U_{i},});
directive (group="fir.V");
VSR=sregtaps(V1[15:6],taps,replicate(i=0; i<=taps; i=i+1)
{V_{i},});
directive (group="fir.Y");
Y=replicate (i=0; i<taps; i=i+1) {Y_{i+1}*YC{i}+} 0;
directive (group="fir.U");
U=replicate (i=0; i<taps; i=i+1) {U_{i+1}*YC{i}+} 0;
directive (group="fir.V");
V=replicate (i=0; i<taps; i=i+1) {V_{i+1}*YC{i}+} 0;

showgroup("*.Y");
showgroup("*.U");
showgroup("*.V");
showgroup("fir.*");
showgroup("matrix.*");
endmodule
```

---

string, string constant

string

Declares a string variable or creates a string

Syntax

```
string [global] <variable list>
```

Arguments

<variable list>

<assigned variable> [, <assigned variable>]*

<assigned variable>

<variable> [= <string expression>]

Syntax

```
<string> = string (<arg list>);
```

Arguments

<arg list>

<expression> [, <expression>]

Directives

None

Description

The `string` function is the character-string counterpart of an integer. It declares and optionally initializes a string variable. In addition, the `string` function can be used to concatenate variables of different types to create a character string.

The rules for interpreting the arguments to the string function are the same as those for interpreting the `error` or the `info` functions: If a nonsignal variable or expression is encountered, its value is used; otherwise, the original name of the signal is used.

Strings support the operators shown in Table 2-17.

Table 2-17 String Operators

Symbol	Name	Result
( )	Expression grouping	String or integer, depending on the expression within ()
+	Concatenate	String
==	Equality compare	Integer
!=	Inequality compare	Integer

Strings and nonstrings cannot be mixed in operators such as + and ==.

```
string x = y + z; //allowed if y and z are strings
integer x = y == "a string"; //allowed if y is a string
string x = y - z; //not allowed
```

A global string variable can be accessed throughout the entire design.

Examples

```
string x, y, x; // declare string variables
string xyz = "a string"; // declare and initialize a variable
string x;
integer y = ...
input X;
x = string(X, "_", y);
// create a string that is the concatenation of
// X , "_", and the integer value y
```

The string function that is supported is `formatStr(x)`, which returns the format of a signal or constant x, as shown in the following example. For more information, see “formatStr” on page 2-95.

```
wire [8] X;
string xxx = formatStr(X); // xxx="unsigned"
```

---

width, log2

width

Width integer function

log2

Bit-width function

Syntax

```
<integer> = width ( <operand> );
```

Argument

<operand>

<signal> | <integer expression>

Syntax

```
<integer> = log2(<operand>);
```

Arguments

<operand>

<integer expression>

Directives

None

Description

The `width` function returns the bit-width of its argument.

If the argument is a signal and it does not have a width, 0 is returned. If the argument is an integer, the `width` function returns the minimum number of bits required to store the integer, unless a larger width is specified.

The number of bits required to store an integer quantity is computed with the assumption that negative integers are signed and positive integers are unsigned.

The `log2` function returns the ceiling of the base-2 log of the given integer, which must be greater than 0. Typically, the log2(n) function computes how many address bits are needed to select one of *n* elements. log2(n) = width(n –1) for n > 0.

Table 2-18 shows examples of log2(n) and width(n) results for different input values of *n*.

Table 2-18 log2(n), width(n)

n	log2(n)	width(n)
9	4	4
8	3	4
7	3	3
4	2	3
3	2	2
–3	NA	3
–4	NA	3
–7	NA	4
–8	NA	4
–9	NA	5

Examples

The following is a simple example of the `width` function:

```
function foo (Z,X);
input X;
output Z;
integer widthX=width(X);
```

The following example shows:

```
wire [8] X;
integer a=10;
integer b=40'd10;
integer c=-40'd10;
integer d=-10;
info (
    "width(X)=",width(X),"\n",
    "width(a)=",width(a),"\n",
    "width(b)=",width(b),"\n",
    "width(c)=",width(c),"\n",
    "width(d)=",width(d),"\n"
);

// generates the following
width(X)= 8
width(a)= 4
width(b)=40
width(c)=40
width(d)= 5
```

3

Directives

This chapter defines directives available in Module Compiler and provides examples of their usage. Directives provide a mechanism for providing operating information to Module Compiler by setting the value of attributes.

Syntax

```
directive [local] (<name> = <value> [, <name> = <value>]*);

directive(<name>);
```

Arguments

<name>

Attribute name

<value>

Valid value for the attribute

Description

Directives provide operating information to Module Compiler by setting the value of attributes. Directives are variables that accept a certain range of integers or certain strings.

Generally, the attributes influence the way a design description is compiled and synthesized, rather than changing the functionality of the design. There are two principal directive types: *local* and *default*. The directive types are defined as follows:

- *Local directives* affect only the next statement. If the next statement is a function call, the entire function call and all functions called from that function are affected.
- A *default directive* issued in a function affects subsequent statements in the same function and in subsequent functions called from the function containing the directive, but it cannot affect statements in the function or module that called the function that contains the directive.

Note:

The global directive is no longer supported, beginning with the 2003.06-SP1 release. The default and global directives now have the same scope.

Table 3-1 lists the set of attributes that Module Compiler recognizes.

Table 3-1 Attributes Recognized by Module Compiler

Attribute	Description	Default
archopt	Controls whether the ripple adder optimizes for speed or area	auto
archtype	Controls whether the architecture of the ripple adder is inverting or noninverting	auto
async_clear	Sets the signal specified by the string value to the flip-flop clear pin	<none>
async_preset	Sets the signal specified by the string value to the flip-flop preset pin	<none>
carrysave	Enables or disables carry-save generation: on, off, convert, or optimize	off
clock	Sets the name of the current clock	CLK
col	Specifies the column position of an instance in its physical function for gate-level tiling	0
dcopt	Enables or disables optimization by Design Compiler: on or off	on
delay	Sets current delay goal, in ps	speed (this value means 1 ps)
delstate	Controls pipeline loaning	0
direxT	Sets the direct sign-extension mode for sum operation: on or off	off
fadelay	Sets final adder delay, only for csa or clsa	<current delay goal>
fatype	Sets final adder architecture: aofcla, cla, clsa, csa, fastcla, ripple, ripple_alt, or auto	auto
group	Sets the current group name	"misc"

Table 3-1 Attributes Recognized by Module Compiler (Continued)

Attribute	Description	Default
indelay	Sets the input operand arrival time, in ps	0
inload	Sets the input operand maximum load, in 0.1 standard load	400
intround	Specifies bit position for internal rounding, or 0 for no rounding	0
logopt	Sets the logic optimization mode: on or off	on
maxtreedepth	Sets the Wallace tree maximum depth: 3 => serial,< large value> => parallel	9999
modname	Sets the module name to the string value provided	<from module declaration>
multtype	Sets the multiplier architecture: Booth, non-Booth, or auto	auto
muxtype	Sets the MUX architecture: andor, mux, or three-state	mux
outdelay	Sets the output operand delay, in ps	0
outload	Sets the output operand load, in 0.1 standard load	30
physical	Enables and disables gate-level tiling: on or off	off
physicalfunction	Names the bit-sliced relative placement block in Module Compiler Language	<none>
pipeline	Sets the pipelining mode: on or off	off
pipeslack	Controls the latency of a design when auto-pipelining cannot meet the delay or latency constraint	0

Table 3-1 Attributes Recognized by Module Compiler (Continued)

Attribute	Description	Default
pipestall	Sets the name of the stall signal	<none>
round	Specifies rounding sum position, or 0 for no rounding	0
rpgen	Enables the ability to structure portions of a design	on
row	Specifies the row position of an instance in its physical function for gate-level tiling	0
scan	Sets the scan test mode: on or off	off
selectop	Sets the optimization mode for MUXs and shifters: msb, lsb, or auto	auto
use_for_size_only	Sets technology library cells as don't use for Module Compiler synthesis and optimization	–

Information on some of these attributes follows Example 3-1.

Example

Example 3-1 directive

```
directive (pipeline = "on"); // set pipeline attribute

directive (pipeline = "on", delay = (x + y) / 2);
// set pipeline and delay attributes
// x and y assumed integers

if (directive(pipeline) == "on") then...// query a directive
```

archopt

The `archopt` attribute controls whether the ripple adder architecture optimizes for speed or for area. The attribute is in effect until reset in a subsequent directive.

You can set `archopt` to the following values:

- `speed` maps to a speed-optimized ripple adder cell selection.
- `size` maps to an area-optimized ripple adder cell selection.
- `auto` maps to the specified design goal. By default, the design goal is set to speed, unless you set the `dp_opt` Module Compiler environment variable to `size`.
- `none` provides backward compatibility. This setting is used with `archtype` set to `inverting` or `noninverting`.

archtype

The `archtype` attribute controls whether the ripple adder architecture is inverting or noninverting. The attribute is in effect until reset in a subsequent directive.

You can set `archtype` to the following values:

- `auto` maps to the best synthesized architecture before logic optimization, based on the `archopt` attribute setting.
- `inverting` maps to the existing `fatype=ripple` attribute setting, providing backward compatibility. This setting is used with `archopt` set to `none`.

- `noninverting` maps to the existing `fatype=ripple_alt` attribute setting, providing backward compatibility. This setting is used with `archopt` set to `none`.

async_preset and async_clear

The `async_preset` attribute specifies the signal connected to the preset input of the flip-flop. This attribute can take the following string value:

- `async_preset = "<signal_connected_to_preset>"`

The `async_clear` attribute specifies the signal connected to the clear input of the flip-flop. This attribute can take the following string value:

- `async_clear = "<signal_connected_to_clear>"`

The default value for both `async_preset` and `async_clear` is `none`.

carrysave

The `carrysave` attribute controls carry-save operand generation. If it is set to `off`, the result of all sum-based operators is true binary. If it is set to `on`, the result is left in carry-save format (the redundant binary format).

Set this attribute to `optimize` to minimize the computational burden of the following addition.

You can also set this attribute to `convert`, which generates carry-save signals that can be passed into the `csconvert` function. However, the use of the `csconvert` function is recommended only for experienced Module Compiler datapath designers.

clock

The `clock` attribute sets the name of the current clock. Module Compiler supports multiple clocks in a design, although each group can have only one clock. Note that whenever you change the current clock, you must also change the group.

col

The `col` attribute specifies the column position of an instance in its physical function for gate-level tiling. The default value for `col` is `0`.

dcopt

Setting the `dcopt` attribute allows you to optimize all, some, or none of your circuit. This is a Boolean attribute that enables Design Compiler optimization when set to `on` and disables optimization when set to `off`. The entire circuit is sent to Design Compiler, but Design Compiler does not touch any instance that was created when `dcopt` was off. Design Compiler optimization is `on` by default.

Note that this feature works only when you are running Design Compiler from Module Compiler. The selective optimization automatically sets the `dont_touch` attribute on the cells using a postprocessing script.

delay

The `delay` attribute sets the current delay goal, in ps. The current delay is tied to the `group` attribute. You cannot change the delay goal without changing `group` as well.

Module Compiler always tries to minimize delay. If minimizing delay is not acceptable, logic optimization should be turned off for that region by use of the `logopt` attribute.

delstate

The `delstate` attribute sets the degree of pipeline loaning. If you enable pipeline loaning, Module Compiler can convert some of the available state registers into pipelines that are used later to improve performance without increasing latency.

The degree of loaning specifies how many state registers can be converted. When `delstate` is set to `0`, Module Compiler does not employ pipeline loaning.

direxT

The `direxT` attribute forces direct sign extension in sum-based operations. If the attribute is set to `off`, Module Compiler performs sign extension by adding a constant value instead of by replicating the sign bit. Direct sign extension should be used for adding or subtracting two operands for different widths.

fadelay

The `fadelay` attribute sets the delay of the final adder if the final adder type is csa or clsa. By default, the final adder is built on the assumption that the output of the adder must arrive before the delay goal, ignoring any circuits at the adder output. This assumption is often wrong in complex circuits; you can use `fadelay` to override this assumption.

fatype

The `fatype` attribute sets the architecture type for the final adder in sum-based operations. The following values are supported:

- `auto`

  This setting (the default) allows Module Compiler to choose an adder to fit the context. As shown in Table 3-2, the `auto` choice depends on the pipeline and optimization of criterion settings.

Table 3-2 Adder Choices

Condition	fatype
pipeline = on	cla
pipeline = off and optimizing criterion is speed	fastcla
pipeline = off and optimizing criterion is not speed	clsa

- `aofcla`

  The aofcla adder is a 1-to-1,024-bit carry-propagate microarchitecture. Compared to fast adders such as fastcla, aofcla reduces area with little or no increase in timing.

In general, the aofcla adder offers 20 to 40 percent reduction in area, compared to the fastcla adder; is comparable or superior in performance to the fastcla and clsa adders; and, for lower bit-widths, is superior in both area and timing to the fastcla adder. However, performance and area depend on the specific design and its library.

- `cla`

  The cla architecture implements a carry-lookahead adder. It uses a sparse carry tree that roughly doubles the delay in the carry tree relative to the fastcla, but it provides significant area savings. Because the tree is sparse, there is ample slack on many of the nets, making logic optimization successful for this structure.

- `clsa`

  The clsa architecture implements a carry-lookahead-select adder. It does not pipeline well, but it is the most flexible and automatically creates an architecture ranging from a ripple to a fastcla adder, depending on the desired delay. In general, the clsa architecture is a good choice, particularly with large delay skews.

- `csa`

  The csa architecture implements a carry-select adder, which ideally achieves delay and O($\sqrt{n}$) area. Module Compiler automatically breaks the csa into multiple stages.

- `fastcla`

  The fastcla architecture implements a fast carry-lookahead adder. It is usually the fastest architecture and also the largest. This architecture is very balanced and tends to improve only minimally during logic optimization.

- `ripple`

  The ripple architecture implements a ripple adder. It is a small, slow adder architecture that is useful in noncritical portions of the design. This architecture produces speed-efficient ripple adders. It maps to an alternating polarity chain of full adders with inverted carry-ins and carry-outs.

- `ripple_alt`

  The ripple_alt adder, another small, slow structure, is the most useful for generating area-efficient ripple adders. It uses only simple full adders.

The ripple and ripple_alt adders are technology independent and are the most useful for noncritical portions of the design. Module Compiler can automatically choose the ripple adder that best satisfies design constraints. For more information, see the *Module Compiler User Guide* section about optimized ripple architecture.

group

The `group` attribute associates a name with a set of operands that belong to the same timing group. This attribute is related to the `delay` attribute. If the delay is changed, the group should be changed as well.

A group consists of one or more operands selected by the designer. Statistics such as area, power, and critical path are maintained for each group and are listed in the design report.

An unlimited number of groups are allowed. The group named `misc`, predefined at the beginning of every design, contains all operands not included in a user-defined group. All operands in a group must have the same delay goal.

indelay

The `indelay` attribute specifies the arrival time of a module input. If `indelay` is positive, it indicates an input arriving later than the default (`0`), making paths from that input more critical. Any negative values are treated as minus infinity, making all paths from that point noncritical.

It is possible to specify different delays for different inputs by interleaving the directive statements with the input declarations. Input delays are specified in ps.

inload

The `inload` attribute specifies the maximum loading allowed on a module input. Module Compiler does not put more than this load value on the inputs that are declared after this directive statement. It is possible to specify different loading for different inputs by interleaving the `directive` statements with the input declarations. The input loading is always specified in 0.1 standard load.

intround

The `intround` attribute makes tradeoffs between area and precision in arithmetic expressions. By default, the value of this attribute is `0`. Increasing the value of `intround` increases the number of bits discarded from each addend in the arithmetic expression.

Because the addends are rounded before being summed, savings in area occur. There are also some small performance improvements. Module Compiler outputs the correct behavioral- and gate-level netlist for each value of `intround`. You can use these files to verify system performance.

logopt

The `logopt` attribute turns logic optimization on or off for specific regions of a design. Turning logic optimization off can be useful in cases where reducing the delay is not desirable or the structure is not understood by Module Compiler.

maxtreedepth

The `maxtreedepth` attribute limits the depth of the Wallace tree used to implement the sum-based functions. The smaller the value of `maxtreedepth`, the more serial and slower the implementation.

The proper use of this attribute allows you to effectively create a serial connection of Wallace trees without changing the network description. The minimum value for this attribute is 3.

Setting the `maxtreedepth` attribute value to 2 and the `multtype` attribute value to `Booth` results in an array multiplier. The `rp_pushpp` Module Compiler environment variable controls the shape of the array multiplier (rectangle or parallelogram).

Setting the `maxtreedepth` attribute value to 9999 and the `multtype` attribute value to `non-Booth` results in a Wallace tree multiplier. The `rp_wallace` Module Compiler environment variable controls the shape of the Wallace tree multiplier (default, alternative).

modname

Normally, the name of the cell to be generated by Module Compiler is the same as the name of the module. The `modname` attribute specifies an alternative name for the cell. This name is also used as the root name of all output files that are unique to the design.

multtype

The `multtype` attribute specifies the multiplier architecture. This attribute works with the `maxtreedepth` attribute to specify the resulting multiplier (array, Wallace tree).The following type values for multiplier architectures are supported:

- `auto`

  The selection of the multiplier is automatic. The Booth architecture is employed if the X and Y inputs have at least 16 bits combined; otherwise, non-Booth architecture is used.

- `non-Booth`

  The non-Booth multiplier generates addends, using only simple logic: inverters for buffering, NOR gates for the basic partial product generators, and OR gates for the sign bits of the partial product generators.

  This type of multiplier generates N partial products of M bits each, where N is the width of the Y input and M is the width of the X input. The non-Booth multiplier is relatively efficient when N and M are small numbers.

- `Booth`

  The Booth multiplier utilizes special library cells to encode the Y inputs and generate the partial products. The number and width of the partial products are summarized in Table 3-3 and Table 3-4.

  This multiplier is the most efficient for signed X and Y and, in particular, signed and even Y. This multiplier is the most efficient when Y is wide.

Table 3-3 Booth, Number Partial Products

Y input with width N	**Number of partial products**
Signed-even N	N/2
Signed-odd N	(N+1)/2
Unsigned-even N	N/2+1 [1]
Unsigned-odd N	(N+1)/2*

1. One partial product is simple (NOR-gate-based).

Table 3-4 Booth, Width Partial Products

X input with width M	Width of partial products
Signed	M+1
Unsigned	M+2

muxtype

The `muxtype` attribute specifies the MUX architecture. It can be set to one of the following:

- `mux`

  The mux architecture is often the best choice and also the most straightforward, because it provides good speed and area. It cannot take advantage of skewed data input arrival times but does optimize the structure when the select inputs have skewed arrival times.

  If the select space is not full, meaning that fewer than 2n data inputs are provided for an n-bit select input, the unused select values are assumed to be don't care values and are used to minimize the area. The behavioral model outputs X if unspecified select values are used, whereas the logic model outputs one of the data inputs.

- `andor`

  The andor architecture is fully timing driven and can provide good performance for highly skewed input arrival times. It is, however, generally larger than the MUX-based implementations and is

also typically slower for inputs with no arrival time skew. If the select space is not full and if an undefined select value is used, the output is 0.

- `tristate`

  The three-state (tristate) architecture uses the decoded select inputs to enable a three-state driver for the selected input onto the output bus. The logic optimizer cannot optimize the three-state buffers, and the increasing load at the output tends to limit the usefulness of this structure.

outdelay

The `outdelay` attribute specifies any external path delays associated with the output of a module. These delays are in the circuit following the cell synthesized by Module Compiler and are added to the Module Compiler path delay.

Thus, greater output delays result in greater net criticality. Negative output delays are not allowed. All delays have units of ps. It is possible to specify different delays for different outputs by interleaving the directive statements with the output declarations.

outload

The `outload` attribute specifies the load associated with an output of a module. This load is placed on the driver of the output. All load values have units of 0.1 standard load. It is possible to specify different loading for different outputs by interleaving the directive statements with the output declarations.

physical

The `physical` attribute enables and disables gate-level tiling. By default, the value of `physical` is `off`.

physicalfunction

The `physicalfunction` attribute allows you to control gate-level tiling. Use `physicalfunction` to create a structured placement block and specify cell row and column positions in that block.

pipeline

The `pipeline` attribute sets automatic pipeline control. When it is set to `on`, Module Compiler automatically creates pipeline stages to meet the current delay goal. When it is set to `off`, no automatic pipeline is inserted (except with pipeline loaning, which inserts the loaned pipeline stages automatically even if pipelining is not enabled).

Manual pipelining and automatic latency deskewing are not affected by this attribute. If you manually insert a pipeline stage that creates latency skews, pipelines are inserted to deskew the latencies when needed, regardless of the current setting of the `pipeline` attribute. You can also use the `ResolveLatency` and the `ResolveLatencyLoop` functions to force a pipelined section of code to a desired latency level.

pipeslack

Use the `pipeslack` attribute to control the latency of a design when auto-pipelining cannot meet the delay or latency constraint. Pipeslack changes the timing goal (delay) of the circuit during pipelining (which is done during the synthesis step).

The new delay goal during pipelining is `delaypipe`. For example:

```
delaypipe = delay - pipeslack
```

During the optimization step, the delay goal reverts back to the original delay set for the design:

```
delayopt = delay
```

The default value of `pipeslack` is `0`.

pipestall

When set to the name of the stall control signal, the `pipestall` attribute stalls all synthesized flip-flops (whether they are state or pipeline registers) when the stall control signal is low.

round

The `round` attribute rounds the result of a sum to the *n*th bit. After the rounding, bit *n* is the new LSB of the output. Rounding to bit 0 is the same as no rounding.

rpgen

Use the `rpgen` attribute to structure a portion of your design (`rpgen="on"`). The default value of `rpgen` is `on`. Therefore, you must set the attribute to `off` at the beginning of the code. The code immediately following the `rpgen="on"` attribute is structured until the `rpgen` attribute is disabled (`rpgen="off"`). The structured portion of the design is preserved through physical synthesis and optimization.

row

The `row` attribute specifies the row position of an instance in its physical function for gate-level tiling. The default value of `row` is `0`.

scan

The `scan` attribute controls whether flip-flops convert into their scan counterparts. When `scan` is set to `on`, the conversion takes place and Module Compiler builds the circuit with good area and delay estimates. When `scan` is set to `off`, no conversion takes place. During report generation, the scan flip-flops are converted back to the original cells.

selectop

The `selectop` attribute controls the ordering of select signals for shifters, rotators, and MUX-based multiplexers. This attribute has three settings:

- When `selectop` is set to `msb`, the select inputs are ordered from MSB to LSB, where the delay from the LSB is the least and the delay from the MSB is the greatest.
- When `selectop` is set to `lsb`, the order is reversed, so the delay from the LSB is the greatest.
- When `selectop` is set to `auto`, Module Compiler orders the select inputs to minimize the delay from the select inputs to the output, based on the select input arrival times.

use_for_size_only

The `use_for_size_only` attribute sets technology-library cells as don't use for Module Compiler synthesis and optimization. When Module Compiler reads the technology library, it detects all cells that have this attribute.

These cells are included in the Don't Use section of the Module Compiler Library Report. For an example of this section, see the *Module Compiler User Guide*.

If the cells with the `use_for_size_only` attribute are instantiated in any Module Compiler Language code, Module Compiler preserves any instances of these cells through optimization and they appear in the Module Compiler-generated netlist.

4

Generic Functions

This chapter includes the following sections:

- Required and Optional Cells
- Generic Functions Listed Alphabetically

Required and Optional Cells

The quality and variety of cells in your library affect the quality of results Module Compiler generates. Module Compiler requires only a minimum set of cells (basic cells) for building a design. However, additional cells, if provided in your library, can contribute to a higher quality of results.

These additional cells are described in Table 4-1. In this table, the cell names are Module Compiler generic cell library names that you might name differently in your own cell library. Note that in naming your own cells there cannot be a leading period "." in the cell name. This limitation is for the first character only. A period can occur after the first letter of the cell name.

Module Compiler has the ability to build pseudocells—critical cells that are not present in your library. This allows you to build structures requiring cells your library lacks. You can construct all cells—excluding the required basic cells, basic D flip-flops, latches, and three-state buffers—as pseudocells. However, properly designed and implemented native cells provide advantages in area, delay, power, and routability over pseudocells.

Important:

When loading technology libraries, it is important that you list the libraries in the order that Module Compiler expects. You must specify the technology file containing the smallest inverter in the library *first*. Module Compiler defines the technology library with the smallest inverter as the *main library*. If you do not list the main library first, Module Compiler might exit abnormally and fail to build the pseudocell library, or it might build a suboptimal circuit.

Because of its direct design style, detailed reporting, and fast synthesis capability, Module Compiler is an excellent library analysis tool, allowing you to build a variety of designs in a short period of time across multiple variants of your library. This can help you quickly analyze the impact of including or excluding a given cell set and tune your library for designs with high datapath content.

In Table 4-1, *Required* means that your library must have these cells. Module Compiler cannot substitute pseudocells for these cells. Without these cells, Module Compiler cannot build the indicated structures. *Optional* means that these cells are optional but recommended. Module Compiler can build pseudocells as substitutes for these cells. If you do provide these cells as native cells in your library, users can get improved results for the indicated structures.

Table 4-1 Required and Optional Cells, Grouped by Class

Class	Used for	Required/ optional	Cell name	Cell equation
Basic cells	All structures	Required	mcgen_inv1a	Y = ~A
		Required	mcgen_nand2a	Y = ~(A & B)
			or	
			mcgen_and2a	Y = A & B
		Required	mcgen_or2a	Y = A | B
			or	
			mcgen_nor2a	Y = ~(A | B)
		Optional	mcgen_buf1a	Y = A
			or	
			mcgen_buf2a	Y = A
		Optional	mcgen_mx2a	Y = S ? D1 : D0
		Optional	mcgen_xnor2a	Y = ~(A ^ B)
			or	
			mcgen_xor2a	Y = A ^ B
MUX and three-state cells	Three-state-based MUXs	Required	tri1a	Noninverting internal three-state buffer
	MUX-based multi-plexers, shifters, and rotators	Optional	mcgen_mx2d	Y = ~(S ? D1 : D0)
			mcgen_mx3a	Y = S1 ? D2 : (S0 ? D1 : D0)
			mcgen_mx4a	Y = S1 ? (S0 ? D3 : D2) : (S0 ? D1 : D0)

Table 4-1 Required and Optional Cells, Grouped by Class (Continued)

Class	Used for	Required/ optional	Cell name	Cell equation
Flip-flops	Sequential designs	Required	fd1a	D flip-flop
			fde1a	D flip-flop with enable
	Scan test mode	Optional	fd1c	D flip-flop, output inverted
			fde1	D flip-flop with enable, output inverted
		Required	fdm1a	Scan flip-flop
			fdem1a	Scan flip-flop with enable
		Optional	fdm1c	Scan flip-flop, output inverted
			fdem1c	Scan flip-flop with enable, output inverted
Latches	Latches and netlist memories	Required	ld1a or	D latch
			ld1b	D latch with enable
		Optional	ld1c	D latch, output inverted
AND-OR, XOR	Functions based on AND-OR and XOR trees	Optional	and3a-and8a	3-to-8-input and
			nand3a-nand8a	3-to-8-input nand
			nor3a-nor8a	3-to-8-input nor
			or3a-or8a	3-to-8-input or
			mcgen_xor3a	Y = A ^ B ^ C
			mcgen_xnor3a	Y = ~(A ^ B ^ C)
Adder cells	Adders	Optional	mcgen_fa1a	S = A ^ B ^ Cin Cout = (A & B) | (A & Cin) | (B & Cin)
			mcgen_ha1a	S = A ^ B Cout = (A & B)
			mcgen_fa2a	S = A ^ B ^ Cin Cout = ~ (A & B) | (A & Cin) | (B & Cin))

Table 4-1 Required and Optional Cells, Grouped by Class (Continued)

Class	Used for	Required/ optional	Cell name	Cell equation
			mcgen_fa1b	S = A ^ B ^ ~Cin Cout = (A & B) | (A & ~Cin) | (B & ~Cin)
			facs2a	S = CSIN ? (A ^ B ^ Cin1): (A ^ B ^ Cin0) Cout0 = ~((A & B) | (A & Cin0) | (B &Cin0)) Cout1 = ~((A & B) | (A & Cin1) | (B & Cin1))
			facs1b	S = CSIN ? (A ^ B ^ ~Cin1) : (A ^ B ^ ~Cin0) Cout0 = (A & B) | (A & ~Cin0) | (B & ~Cin0) Cout1 = (A & B) | (A & ~Cin1) | (B & ~Cin1)
			facs3a	S = CSIN ^ A ^ B Cout0 = ~(A & B) Cout1 = ~(A | B)
			facs4a	S = ~CSIN ^ A ^ B Cout0 = ~(A & B) Cout1 = ~(A | B)
			mcgen_mx2d	Y = ~(S ? D1 : D0)
			mcgen_ao1f	Y = (~A & ~B) | ~C
			mcgen_oa1f	Y = (~A | ~B) & ~C

Table 4-1 Required and Optional Cells, Grouped by Class (Continued)

Class	Used for	Required/ optional	Cell name	Cell equation
Adder cells	Compar-ators and incre-mentors	Optional	hacs2a	S = CSIN ? A ^ B : A Cout = ~(A & B)
			hacs1b	CSIN ? A ^ ~B : A Cout = A & ~(B)
			mcgen_ha2a	S = A ^ B Cout = ~(A & B)
			mcgen_ha1b	S = A ^ ~(B) Cout = A & ~(B)
			faccs1b	Cout1 = (A & B) | (A & ~Cin1) | (B &~Cin1) Cout0 = (A & B) | (A & ~Cin0) | (B &~Cin0)
Multi plier cells	Booth-encoded multipliers	Optional	mule2a	S = ~(A ^ B) M = C | ~A & ~B Z = ~C | A & B
			mulpa1b	P = (~(S & X1 | ~S & X0)) & ~M | ~(~(S & X1 | ~S & X0)) & ~Z Booth partial product generation
			mulpa2	P = ~((~(S & X1 | ~S & X0)) &~M | ~(~(S & X1 | ~S & X0)) & ~Z) Booth partial product generation

Generic Functions Listed Alphabetically

The Module Compiler generic function library is a collection of low-level functions. Each function is linked to a corresponding technology-specific cell if one exists, regardless of the cell and pin names the vendor uses.

All Module Compiler generic functions begin with the mcgen_ prefix, which keeps generic function names unique and prevents name clashes with existing technology cell names.

Using Module Compiler version 2000.05 or later, you might get an error message (undefined functions) when you synthesize Module Compiler Language code having generic function instantiations. If this occurs, modify your Module Compiler Language code to change the names of the instantiated generic functions to the new generic function names (those having a mcgen_ prefix).

If there is no corresponding cell in the technology library, Module Compiler synthesizes the generic cell function from two or more technology-specific cells.

Table 4-2 shows the generic functions available in Module Compiler. If the value in the Bus column is Y, the function accepts bused inputs and outputs; otherwise, it does not.

Table 4-2 Generic Functions Listed Alphabetically

New generic function name	Bus	Cell equation
mcgen_and2a (Y,A,B)	Y	Y = A & B
mcgen_and2b (Y,A,B)	Y	Y = ~A & B
mcgen_and2c (Y,A,B)	Y	Y = ~A & ~B

Table 4-2 Generic Functions Listed Alphabetically (Continued)

New generic function name	Bus	Cell equation
mcgen_and3a (Y,A,B,C)	Y	Y = A & B & C
mcgen_and3b (Y,A,B,C)	Y	Y = ~A & B & C
mcgen_and3c (Y,A,B,C)	Y	Y = ~A & ~B & C
mcgen_and3d (Y,A,B,C)	Y	Y = ~A & ~B & ~C
mcgen_and4a (Y,A,B,C,D)	Y	Y = A & B & C & D
mcgen_and4b (Y,A,B,C,D)	Y	Y = ~A & B & C & D
mcgen_and4c (Y,A,B,C,D)	Y	Y = ~A & ~B & C & D
mcgen_and4d (Y,A,B,C,D)	Y	Y = ~A & ~B & ~C & D
mcgen_and4e (Y,A,B,C,D)	Y	Y = ~A & ~B & ~C & ~D
mcgen_and5a (Y,A,B,C,D,E)	Y	Y = A & B & C & D & E
mcgen_and5b (Y,A,B,C,D,E)	Y	Y = ~A & B & C & D & E
mcgen_and5c (Y,A,B,C,D,E)	Y	Y = ~A & ~B & C & D & E
mcgen_and5d (Y,A,B,C,D,E)	Y	Y = ~A & ~B & ~C & D & E
mcgen_and5e (Y,A,B,C,D,E)	Y	Y = ~A & ~B & ~C & ~D & E
mcgen_and5f (Y,A,B,C,D,E)	Y	Y = ~A & ~B & ~C & ~D & ~E
mcgen_and6a (Y,A,B,C,D,E,F)	Y	Y = A & B & C & D & E & F
mcgen_and6g (Y,A,B,C,D,E,F)	Y	Y = ~A & ~B & ~C & ~D & ~E & ~F
mcgen_and8a (Y,A,B,C,D,E,F,G,H)	Y	Y = A & B & C & D & E & F & G & H
mcgen_and8i (Y,A,B,C,D,E,F,G,H)	Y	Y = ~A & ~B & ~C & ~D & ~E &~F & ~G & ~H
mcgen_ao1a (Y,A,B,C)	Y	Y = (A & B) | C
mcgen_ao1b (Y,A,B,C)	Y	Y = (~A&B) | C

Table 4-2 Generic Functions Listed Alphabetically (Continued)

New generic function name	Bus	Cell equation
mcgen_ao1c (Y,A,B,C)	Y	Y = (~A & ~B) | C
mcgen_ao1d (Y,A,B,C)	Y	Y = (A & B) | ~C
mcgen_ao1e (Y,A,B,C)	Y	Y = (~A & B) | ~C
mcgen_ao1f (Y,A,B,C)	Y	Y = (~A & ~B) | ~C
mcgen_ao2a (Y,A,B,C,D)	Y	Y = (A & B) | C | D
mcgen_ao2b (Y,A,B,C,D)	Y	Y = (~A & B) | C | D
mcgen_ao2c (Y,A,B,C,D)	Y	Y = (~A & ~B) | C | D
mcgen_ao2d (Y,A,B,C,D)	Y	Y = (A & B) | ~C | D
mcgen_ao2e (Y,A,B,C,D)	Y	Y = (~A & B) | ~C | D
mcgen_ao2f (Y,A,B,C,D)	Y	Y = (~A & ~B) | ~C | D
mcgen_ao2g (Y,A,B,C,D)	Y	Y = (A & B) | ~C | ~D
mcgen_ao2h (Y,A,B,C,D)	Y	Y = (~A & B) | ~C | ~D
mcgen_ao2i (Y,A,B,C,D)	Y	Y = (~A & ~B) | ~C | ~D
mcgen_ao3a (Y,A,B,C,D)	Y	Y = (A & B & C) | D
mcgen_ao3b (Y,A,B,C,D)	Y	Y = (~A & B & C) | D
mcgen_ao3c (Y,A,B,C,D)	Y	Y = (~A & ~B & C) | D
mcgen_ao3d (Y,A,B,C,D)	Y	Y = (~A & ~B & ~C) | D
mcgen_ao3e (Y,A,B,C,D)	Y	Y = (A & B & C) | ~D
mcgen_ao3f (Y,A,B,C,D)	Y	Y = (~A & B & C) | ~D
mcgen_ao3g (Y,A,B,C,D)	Y	Y = (~A & ~B & C) | ~D
mcgen_ao3h (Y,A,B,C,D)	Y	Y = (~A & ~B & ~C) | ~D

Table 4-2 Generic Functions Listed Alphabetically (Continued)

New generic function name	Bus	Cell equation
mcgen_ao4a (Y,A,B,C,D)	Y	Y = (A & B) | (C & D)
mcgen_ao4b (Y,A,B,C,D)	Y	Y = (~A & B) | (C & D)
mcgen_ao4c (Y,A,B,C,D)	Y	Y = (~A & ~B) | (C & D)
mcgen_ao4d (Y,A,B,C,D)	Y	Y = (~A & B) | (~C & D)
mcgen_ao4e (Y,A,B,C,D)	Y	Y = (~A & B) | (~C & ~D)
mcgen_ao4f (Y,A,B,C,D)	Y	Y = (~A & ~B) | (~C & ~D)
mcgen_ao5a (Y,A,B,C,D,E)	Y	Y = (A & B & C) | (D & E)
mcgen_ao5b (Y,A,B,C,D,E)	Y	Y = (~A & B & C) | (D & E)
mcgen_ao6a (Y,A,B,C)	Y	Y = (A & B) | (A & C) | (B & C)
mcgen_ao7a (Y,A,B,C,D,E,F)	Y	Y = (A & B) | (C & D) | (E & F)
mcgen_ao7g (Y,A,B,C,D,E,F)	Y	Y = (~A & ~B) | (~C & ~D) | (~E & ~F)
mcgen_ao8a (Y,A,B,C,D,E)	Y	Y = (A & B) | (C & D) | E
mcgen_ax1a (Y,A,B,C)	Y	Y = (A & B) ^ C
mcgen_buf1a (Y,A)	Y	Y = A
mcgen_buf2a (Y,A)	Y	Y = A
mcgen_fa1a (Cout,S,A,B,Cin)	N	S = A ^ B ^ Cin Cout = A & B | A & Cin | B & Cin
mcgen_fa1b (Cout,S,A,B,Cin)	N	S = A ^ B ^ ~Cin Cout = A & B | A & ~Cin | B & ~Cin
mcgen_fa2a (Cout,S,A,B,Cin)	N	S = A ^ B ^ Cin Cout = ~(A & B | A & Cin | B & Cin)
mcgen_fac1b (Cout,A,B,Cin)	Y	Cout = A & B | A & ~Cin | B & ~Cin
mcgen_fac2a (Cout,A,B,Cin)	Y	Cout = ~(A & B | A & Cin | B & Cin)

Table 4-2 Generic Functions Listed Alphabetically (Continued)

New generic function name	Bus	Cell equation
mcgen_ha1a (Cout,S,A,B)	N	S = A ^ B Cout = A & B
mcgen_ha1b (Cout,S,A,B)	N	S = A ^ (~B) Cout = A & (~B)
mcgen_ha2a (Cout,S,A,B)	N	S = A ^ B Cout = ~(A & B)
mcgen_inv1a (Y,A)	Y	Y = ~A
mcgen_mx2a (Y,D0,D1,S)	Y	Y = S ? D1 : D0
mcgen_mx2d (Y,D0,D1,S)	Y	Y = ~(S ? D1 : D0)
mcgen_mx3a (Y,D0,D1,D2,S0,S1)	Y	Y = S1 ? D2 : (S0 ? D1 : D0)
mcgen_mx4a (Y,D0,D1,D2,D3,S0,S1)	Y	Y = S1 ? (S0 ? D3 : D2) : (S0 ? D1 : D0)
mcgen_mx4e (Y,D0,D1,D2,D3,S0,S1)	Y	Y = ~(S1 ? (S0 ? D3 : D2) : (S0 ? D1 : D0))
mcgen_nand2a (Y,A,B)	Y	Y = ~(A & B)
mcgen_nand2b (Y,A,B)	Y	Y = ~(~A & B)
mcgen_nand2c (Y,A,B)	Y	Y = ~(~A & ~B)
mcgen_nand3a (Y,A,B,C)	Y	Y = ~(A & B & C)
mcgen_nand3b (Y,A,B,C)	Y	Y = ~(~A & B & C)
mcgen_nand3c (Y,A,B,C)	Y	Y = ~(~A & ~B & C)
mcgen_nand3d (Y,A,B,C)	Y	Y = ~(~A & ~B & ~C)
mcgen_nand4a (Y,A,B,C,D)	Y	Y = ~(A & B & C & D)
mcgen_nand4d (Y,A,B,C,D)	Y	Y = ~(~A & ~B & ~C & D)
mcgen_nand4e (Y,A,B,C,D)	Y	Y = ~(~A & ~B & ~C & ~D)
mcgen_nand5a (Y,A,B,C,D,E)	Y	Y = ~(A & B & C & D & E)
mcgen_nand5b (Y,A,B,C,D,E)	Y	Y = ~(~A & B & C & D & E)

Table 4-2 Generic Functions Listed Alphabetically (Continued)

New generic function name	Bus	Cell equation
mcgen_nand5c (Y,A,B,C,D,E)	Y	Y = ~(~A & ~B & C & D & E)
mcgen_nand5d (Y,A,B,C,D,E)	Y	Y = ~(~A & ~B & ~C & D & E)
mcgen_nand5e (Y,A,B,C,D,E)	Y	Y = ~(~A & ~B & ~C & ~D & E)
mcgen_nand5f (Y,A,B,C,D,E)	Y	Y = ~(~A & ~B & ~C & ~D & ~E)
mcgen_nand6a (Y,A,B,C,D,E,F)	Y	Y = ~(A & B & C & D & E & F)
mcgen_nand8a (Y,A,B,C,D,E,F,G,H)	Y	Y = ~(A & B & C & D & E & F & G & H)
mcgen_nor2a (Y,A,B)	Y	Y = ~(A | B)
mcgen_nor2b (Y,A,B)	Y	Y = ~(~A | B)
mcgen_nor2c (Y,A,B)	Y	Y = ~(~A | ~B)
mcgen_nor3a (Y,A,B,C)	Y	Y = ~(A | B | C)
mcgen_nor3b (Y,A,B,C)	Y	Y = ~(~A | B | C)
mcgen_nor3c (Y,A,B,C)	Y	Y = ~ (~A | ~B | C)
mcgen_nor3d (Y,A,B,C)	Y	Y = ~(~A | ~B | ~C)
mcgen_nor4a (Y,A,B,C,D)	Y	Y = ~(A | B | C | D)
mcgen_nor4b (Y,A,B,C,D)	Y	Y = ~(~A | B | C | D)
mcgen_nor4c (Y,A,B,C,D)	Y	Y = ~(~A | ~B | C | D)
mcgen_nor4d (Y,A,B,C,D)	Y	Y = ~(~A | ~B | ~C | D)
mcgen_nor4e (Y,A,B,C,D)	Y	Y = ~(~A | ~B | ~C | ~D)
mcgen_nor5a (Y,A,B,C,D,E)	Y	Y = ~(A | B | C | D | E)
mcgen_nor5b (Y,A,B,C,D,E)	Y	Y = ~(~A | B | C | D | E)
mcgen_nor5c (Y,A,B,C,D,E)	Y	Y = ~(~A | ~B | C | D | E)

Table 4-2 Generic Functions Listed Alphabetically (Continued)

New generic function name	Bus	Cell equation
mcgen_nor5d (Y,A,B,C,D,E)	Y	Y = ~(~A | ~B | ~C | D | E)
mcgen_nor5e (Y,A,B,C,D,E)	Y	Y = ~ (~A | ~B | ~C | ~D | E)
mcgen_nor5f (Y,A,B,C,D,E)	Y	Y = ~(~A | ~B | ~C | ~D | ~E)
mcgen_nor6a (Y,A,B,C,D,E,F)	Y	Y = ~(A | B | C | D | E | F)
mcgen_oa1a (Y,A,B,C)	Y	Y = (A | B) & C
mcgen_oa1b (Y,A,B,C)	Y	Y = (~A | B) & C
mcgen_oa1c (Y,A,B,C)	Y	Y = (~A | ~B) & C
mcgen_oa1d (Y,A,B,C)	Y	Y = (A | B) & ~C
mcgen_oa1e (Y,A,B,C)	Y	Y = (~A | B) & ~C
mcgen_oa1f (Y,A,B,C)	Y	Y = (~A | ~B) & ~C
mcgen_oa2a (Y,A,B,C,D)	Y	Y = (A | B) & C & D
mcgen_oa2b (Y,A,B,C,D)	Y	Y = (~A | B) & C & D
mcgen_oa2c (Y,A,B,C,D)	Y	Y = (~A | ~B) & C & D
mcgen_oa2d (Y,A,B,C,D)	Y	Y = (A | B) & ~C & D
mcgen_oa2e (Y,A,B,C,D)	Y	Y = (~A | B) & ~C & D
mcgen_oa2f (Y,A,B,C,D)	Y	Y = (~A | ~B) & ~C & D
mcgen_oa2g (Y,A,B,C,D)	Y	Y = (A | B) & ~C & ~D
mcgen_oa2h (Y,A,B,C,D)	Y	Y = (~A | B) & ~C & ~D
mcgen_oa2i (Y,A,B,C,D)	Y	Y = (~A |~B) & ~C & ~D
mcgen_oa3a (Y,A,B,C,D)	Y	Y = (A | B | C) & D
mcgen_oa3b (Y,A,B,C,D)	Y	Y = (~A | B | C) & D

Table 4-2 Generic Functions Listed Alphabetically (Continued)

New generic function name	Bus	Cell equation
mcgen_oa3c (Y,A,B,C,D)	Y	Y = (~A | ~B | C) & D
mcgen_oa3d (Y,A,B,C,D)	Y	Y = (~A | ~B | ~C) & D
mcgen_oa3e (Y,A,B,C,D)	Y	Y = (A | B | C) & ~D
mcgen_oa3f (Y,A,B,C,D)	Y	Y= (~A | B | C) & ~D
mcgen_oa3g (Y,A,B,C,D)	Y	Y = (~A | ~B | C) & ~D
mcgen_oa3h (Y,A,B,C,D)	Y	Y = (~A | ~B | ~C) & ~D
mcgen_oa4a (Y,A,B,C,D)	Y	Y = (A | B) & (C | D)
mcgen_oa4b (Y,A,B,C,D)	Y	Y = (~A | B) & (C | D)
mcgen_oa4c (Y,A,B,C,D)	Y	Y = (~A | ~B) & (C | D)
mcgen_oa4d (Y,A,B,C,D)	Y	Y = (~A | B) & (~C | D)
mcgen_oa4e (Y,A,B,C,D)	Y	Y = (~A | B) & (~C | ~D)
mcgen_oa4f (Y,A,B,C,D)	Y	Y = (~A | ~B) & (~C | ~D)
mcgen_oa5a (Y,A,B,C,D,E)	Y	Y = (A | B | C) & (D | E)
mcgen_oa5b (Y,A,B,C,D,E)	Y	Y = (~A | B | C) & (D | E)
mcgen_oa7a (Y,A,B,C,D,E,F)	Y	Y = (A | B) & (C | D) & (E | F)
mcgen_oa7g (Y,A,B,C,D,E,F)	Y	Y = (~A | ~B) & (~C | ~D) & (~E | ~F)
mcgen_oa8a (Y,A,B,C,D,E)	Y	Y = (A | B) & (C | D) & E
mcgen_or2a (Y,A,B)	Y	Y = A | B
mcgen_or2b (Y,A,B)	Y	Y = ~A | B
mcgen_or2c (Y,A,B)	Y	Y = ~A | ~B
mcgen_or3a (Y,A,B,C)	Y	Y = A | B | C

Table 4-2 Generic Functions Listed Alphabetically (Continued)

New generic function name	Bus	Cell equation
mcgen_or3b (Y,A,B,C)	Y	Y = ~A | B | C
mcgen_or3c (Y,A,B,C)	Y	Y = ~A | ~B | C
mcgen_or3d (Y,A,B,C)	Y	Y = ~A | ~B | ~C
mcgen_or4a (Y,A,B,C,D)	Y	Y = A | B | C | D
mcgen_or4b (Y,A,B,C,D)	Y	Y = ~A | B | C | D
mcgen_or4c (Y,A,B,C,D)	Y	Y = ~A | ~B | C | D
mcgen_or4d (Y,A,B,C,D)	Y	Y = ~A | ~B | ~C | D
mcgen_or4e (Y,A,B,C,D)	Y	Y = ~A | ~B | ~C | ~D
mcgen_or5a (Y,A,B,C,D,E)	Y	Y = A | B | C | D | E
mcgen_or5b (Y,A,B,C,D,E)	Y	Y = ~A | B | C | D| E
mcgen_or5c (Y,A,B,C,D,E)	Y	Y = ~A | ~B | C | D | E
mcgen_or5d (Y,A,B,C,D,E)	Y	Y = ~A | ~B | ~C | D | E
mcgen_or5e (Y,A,B,C,D,E)	Y	Y = ~A | ~B | ~C | ~D | E
mcgen_or5f (Y,A,B,C,D,E)	Y	Y = ~A | ~B | ~C | ~D | ~E
mcgen_or6a (Y,A,B,C,D,E,F)	Y	Y = A | B | C | D | E | F
mcgen_or6g (Y,A,B,C,D,E,F)	Y	Y = ~A | ~B | ~C | ~D | ~E | ~F
mcgen_or8i (Y,A,B,C,D,E,F,G,H)	Y	Y = ~A | ~B | ~C | ~D |~ E | ~F |~G | ~H
mcgen_xa1a (Y,A,B,C)	Y	Y = (A ^ B) & C
mcgen_xa1b (Y,A,B,C)	Y	Y = (~A ^ B) & C
mcgen_xa1d (Y,A,B,C)	Y	Y = (~A ^ B) & ~C
mcgen_xnor2a (Y,A,B)	Y	Y = ~(A ^ B)

Table 4-2 Generic Functions Listed Alphabetically (Continued)

New generic function name	Bus	Cell equation
mcgen_xnor3a (Y,A,B,C)	Y	Y = ~(A ^ B ^ C)
mcgen_xor2a (Y,A,B)	Y	Y = A ^ B
mcgen_xor2b (Y,A,B)	Y	Y = ~(A ^ B)
mcgen_xor3a (Y,A,B,C)	Y	Y = A ^ B ^ C
mcgen_xor3b (Y,A,B,C)	Y	Y = ~(A ^ B ^ C)

5

Environment Variables and dc_shell Commands

This chapter includes the following sections:

- Environment Variables
- Command for Starting the Module Compiler GUI
- dc_shell Commands for Running Module Compiler

Environment Variables

You can set Module Compiler environment variables by

- Using the `mcenv` utility
- Using a text editor to modify the mc.env file (you must run Module Compiler once for that file to be automatically created)
- Using command-line switches when you start Module Compiler
- Using the Module Compiler GUI (not all environment variables can be set from the GUI)

The Module Compiler environment variables are stored in the mc.env file. For information about the mc.env file and about using the `mcenv` utility and command-line switches, see “Module Compiler Environment Variables” in Chapter 2 in the *Module Compiler User Guide*.

There are three tables in this section, including environment variables, relative placement environment variables, and reserved environment variables.

Table 5-1 lists Module Compiler environment variables you might want to set, depending on your design goals. Unless otherwise stated, the +|– value indicates + (plus sign) to enable and – (minus sign) to disable.

Table 5-1 Environment Variables

Module Compiler environment variable	Equivalent option	Default	Description
<none>	-pf	<none>	If used (for example, mc -tech abc -pf), the input Module Compiler Language file is parsed before the library files are read; otherwise, the library files are read first and then the input file is parsed.
<none>	-sd	<none>	Scan debug: Scan registers are left in the output netlist; otherwise, they are replaced by nonscan cells.
derate_fast_ named_opcond	<none>	synlibcond	Named opcond for fast operating condition.
derate_slow_ named_opcond	<none>	synlibcond	Named opcond for slow operating condition.
derate_typ_ named_opcond	<none>	synlibcond	Named opcond for typical operating condition.
dp_archopt	-archopt	auto	Controls whether the ripple adder architecture is inverting or noninverting (auto, inverting, noninverting).

Table 5-1 Environment Variables (Continued)

Module Compiler environment variable	Equivalent option	Default	Description
dp_archtype	-archtype	auto	Controls whether the ripple adder architecture optimizes for speed or area (auto, speed, size).
dp_asynchFF_ active_high	<none>	–	If +, selects asynchronous active-high set/reset flip-flops. Otherwise, picks up asynchronous active-low set/reset flip-flops. (+|-)
dp_behav_ mod_name	<none>	<none>	Module name that appears in the behavioral file. The default is taken from the name of the module in the .mcl file.
dp_bit_range_lhs	<none>	–	Sets whether bit range signals are allowed to appear on the left side. (+|-)
dp_bver_name	<none>	(dp_name).bvrl	Behavioral model output name.
dp_bver_out	-b	+	Behavioral model output flag. (+|-)
dp_bvhd_name	<none>	(dp_desdir)(dp_name).bvhd	Name of VHDL behavioral output.
dp_clockgating	-cg	–	Sets the clock-gating capability. (+|-)
dp_contwarn	-cw	+	Continue on warning flag. (+|-)

Table 5-1 Environment Variables (Continued)

Module Compiler environment variable	Equivalent option	Default	Description
dp_db_out	-db	–	Controls whether a .db netlist is generated. (+|–)
dp_dc_behmode	<none>	–	Design Compiler processes behavioral code. (+|–)
dp_dc_checkdesign	<none>	–	Checks design after compiling in Design Compiler. (+|–)
dp_dc_compile	<none>	–	Compiles in Design Compiler. (+|–)
dp_dc_grouprep	<none>	–	Enables analysis for each group in Design Compiler. (+|–)
dp_dc_mapeffort	<none>	med	Sets Design Compiler mapping effort, controlling the quality of results and runtime (low, med, high).
dp_dc_mapincr	<none>	–	Specifies incremental mapping in Design Compiler (doesn't change the circuit drastically). (+|–)
dp_dc_netlist	<none>	(dp_name).dc.vrl	Name of the netlist file written by Design Compiler.
dp_dc_output	<none>	(dp_name).dc.rep	Name of the report file written by Design Compiler.

Table 5-1 Environment Variables (Continued)

Module Compiler environment variable	Equivalent option	Default	Description
dp_dc_run	-dc_run	–	Runs Design Compiler. (+|-)
dp_dc_style_ reg_name	<none>	–	Enables Design Compiler register naming. (+|-)
dp_dc_wireload	<none>	Synlinear2.5	Wire load model name.
dp_debugsim	-debugsim	–	Enables and disables the use of debugging names in the netlist models. (+|-)
dp_default_ clockname	<none>	CLK	Specifies the default clock name to be used for clocking registers.
dp_desdir	<none>	./	Design directory.
dp_edf_out	<none>	–	EDIF output file flag. (+|-)
dp_editor	<none>	vi	Specifies the default text editor (vi or emacs) called by the GUI.
dp_equalglob	-eg	+	Specifies global equalization. (+|-)
dp_equalpass	-ep	1	Sets number of equalization iterations. (1, 2, ...)

Table 5-1 Environment Variables (Continued)

Module Compiler environment variable	Equivalent option	Default	Description
dp_ fileSearchString	<none>	*.mcl	Specifies the search string used in the Module Compiler GUI. By default, Module Compiler searches for files to open that have an .mcl suffix.
dp_graphmode	-gm	+	Enables graphics mode. (+|–)
dp_guilayout_fname	<none>	.dp_layout	Name of GUI layout file.
dp_guilayout_out	<none>	(dp_layout_out)	Flag to generate GUI layout file. (+|–)
dp_in	-i	dp_libpath/test.mcl	Input file name; in GUI mode, the input file name is taken from the session.
dp_include_dir	<none>	(dp_desdir)include/	Include file directory name.
dp_inload	-il	400	Default maximum input load, in 0.1 standard load (any positive real number).
dp_iter_input_fname	-pi	–	Parameter iterator file name (–for no iteration).
dp_keepscan	<none>	–	Controls whether inserted scan cells are retained in the final netlist Module Compiler generates. (+|–)

Table 5-1 Environment Variables (Continued)

Module Compiler environment variable	Equivalent option	Default	Description
dp_lang_out	-lang	verilog	Specifies the simulation language format (Verilog, VHDL, Verilog_VHDL).
dp_layout_fname	<none>	(dp_desdir) (dp_name).layout	Relative placement file name.
dp_layout_out	-layout	+	Generates relative placement layout output. (+|–)
dp_librep_fname	<none>	(dp_tech).rep	Library report file name.
dp_link_library	<none>	–	Sets names of black box files; use quotation marks if specifying multiple black boxes; quotation marks can be omitted if there is only one .db file (–for no black box files).
dp_log	-l	–	Log file name (–for standard output).
dp_logHeight	<none>	25	GUI log window height, in characters.
dp_logmode	-logmode	w	Log file open mode: “a” (append) or “w” (write).
dp_logopt	-opt	–l	Optimization step selection. (–l all)
dp_longname	-ln	–	Enables and disables the use of group names in the netlist models. (+|–)

Table 5-1 Environment Variables (Continued)

Module Compiler environment variable	Equivalent option	Default	Description
dp_lowercase_ inst_name	<none>	–	Generates instance names in lowercase. (+|-)
dp_lowercase_ net_name	<none>	–	Generates net names in lowercase. (+|-)
dp_max_menu_len	<none>	50	Sets the maximum number of entries in library browser menus.
dp_maxerrs	-me	10	Maximum number of similar messages allowed. (1, 2, ...)
dp_maxff	<none>	10000	Maximum number of flip-flops in synthesis status display. (1, 2, ...)
dp_maxgiter	-gi	2	Number of global optimization iterations. (1, 2, ...)
dp_maxlat	<none>	10	Maximum number of latency cycles in synthesis status display. (1, 2, ...)
dp_maxliter	-li	4	Maximum number of local optimization iterations. (1, 2, ...)
dp_maxsect	<none>	100000	Maximum number of sections in synthesis status display.
dp_new_behav_ model	<none>	–	Enables use of ensreg with pipestall. (+|-)

Table 5-1 Environment Variables (Continued)

Module Compiler environment variable	Equivalent option	Default	Description
dp_opcond	-oc	slow	Operating condition (typ, fast, slow).
dp_opt	-o	speed	Optimization mode (speed, size, or delay goal).
dp_outload	-ol	30	Default output load, in 0.1 standard load (any positive real number).
dp_param	-par	–	Comma-separated list of parameters (for example, n=4,m=2); use –for no parameters.
dp_paramlist	<none>	.dp_params	Parameter information file name.
dp_pipe	-p	–	Default pipelining enable flag (+|-).
dp_pipemargin	-ps	0	Default pipeline margin, in ps. (0, 1, ...)
dp_preprocessor	-pp	mcp	Selects parser (mcp for new parser, old for old parser).
dp_prop_fname	<none>	(dp_techdir)/(dp_tech).prop	Name of the property file.
dp_pseudo_flat	-fp	+	Flattens pseudocells. (+|-)
dp_pseudo_opt	<none>	–	Allows pseudocell introduction during optimization. (+|-)

Table 5-1 Environment Variables (Continued)

Module Compiler environment variable	Equivalent option	Default	Description
dp_pseudotechdir	<none>	(dp_techdir)	Name of directory for pseudocells.
dp_ pseudotechglobaldir	<none>	./pcellgloballib	Sets directory for global cache directory for makeMcLibCache pseudocell generation.
dp_ pseudotechlocaldir	<none>	./pcelllocallib	Sets directory for local cache directory for automatic pseudocell generation.
dp_register_input	<none>	–	Enables automatic registering of input. (+|-)
dp_register_output	<none>	–	Enables automatic registering of output. (+|-)
dp_regtrees	-rt	–	Builds regular trees. (+|-)
dp_rep_name	<none>	(dp_name).report	Report output file name.
dp_rep_out	-r	+	Report output file flag. (+|-)
dp_retime	<none>	off	Enables retiming and sets level of retiming effort (off, low, high).
dp_scanmode	-sm	–	Scan test mode flag. (+|-)
dp_scanmode1	<none>	+	Convert scan reg back to D type. (+|-)

Table 5-1 Environment Variables (Continued)

Module Compiler environment variable	Equivalent option	Default	Description
dp_scdf_out	<none>	–	MCDF output file flag. (+|–)
dp_sdc_in	-sdc	–	Passes an SDC constraints file as an argument to Module Compiler (–for no constraints file).
dp_statHeight	<none>	75	Height of the bar graph in the optimization status window.
dp_ statNumStatBars	<none>	45	Number of bars in the optimization bar graphs.
dp_statWidth	<none>	210	Width of the bar graph in the optimization status window.
dp_strict	-strict	+	Strict language usage flag; when enabled, use of obsolete constants, functions, or hidden conversions generates warnings. (+|–)
dp_tbl_name	-t	table	Name of the table file.
dp_tbl_out	-to	+	Generates the table file. (+|–)
dp_tech	-tech	<none>	Specifies the technology library that Module Compiler uses.
dp_tech_lib	<none>	(dp_techdir)/(dp_tech).db	Name of the technology gate library file(s).

Table 5-1 Environment Variables (Continued)

Module Compiler environment variable	Equivalent option	Default	Description
dp_thermHeight	<none>	20	Thermometer height, in pixels, for the synthesis status display.
dp_thermWidth	<none>	200	Thermometer width, in pixels, for the synthesis status display.
dp_toplevel	-tl	–	Chip top-level flag. (+|-)
dp_treetype	<none>	auto	Interconnect model. (auto, best_case_tree, worst_case_tree, or balanced_tree)
dp_userpath	<none>	.	Includes directory search path; this is a colon-separated list of directories; use "." for no path.
dp_ver_name	<none>	(dp_name).vrl	Verilog structural model file name.
dp_ver_out	-v	+	Verilog structural model file flag. (+|-)
dp_verilog_ nonblocking	<none>	+	Flag for forcing Module Compiler to use nonblocking assignments for cycle-based simulators. (+|-)
dp_verilog_ resolution	<none>	10	Verilog simulator resolution. (1, 2, ...)
dp_verilog_sim_ resolution	<none>	10	Verilog simulator resolution, in ps. (1, 2, ...)

Table 5-1 Environment Variables (Continued)

Module Compiler environment variable	Equivalent option	Default	Description
dp_verilog_vhdl_ nonblocking_delay	<none>	1	If set to 0, there is no delay statement for behavioral simulations.
dp_verilog_ vhdl_time_unit	<none>	ns	Controls reference part of time scale.
dp_vhdl_name	<none>	(dp_desdir)(dp_name).vhd	Name of the VHDL gate-level file.
dp_vmode	-m	normal	Verbose reporting mode (normal, verbose, debug).

Table 5-2 lists Module Compiler relative placement environment variables you might want to set, depending on your design goals. Unless otherwise stated, the +|– value indicates + (plus sign) to enable and – (minus sign) to disable.

Table 5-2 Relative Placement Environment Variables

Module Compiler environment variable	Value	Default	Description
dp_physical	+|-	–	Turns relative placement optimization on or off. Top-level master switch. When on, makes the physical submenu viewable in the Module Compiler GUI. (+|–)
dp_layout_out	+|-	+	Turns relative placement layout output on or off. When off, Module Compiler suppresses relative placement output results. The equivalent command-line option is -layout. (+|–)

Table 5-2 Relative Placement Environment Variables (Continued)

Module Compiler environment variable	Value	Default	Description
rp_col_merge	0, 1, or 2	1	Sets the value used for column merging: 0 - None 1 - Normal 2 - Aggressive
rp_grouping	+|-	+	Sets logical grouping to apply to the specified physical grouping in your Module Compiler Language input file. Module Compiler always supports logical grouping. (+|-)
rp_logical_unith	*user_ calculation*	0.0	Sets the value needed for accurate wire length estimates. You must calculate the logical height unit value and enter it.
rp_io_loc	0, 1, or 2	1	Sets the I/O locations: 0 - No input/output ports specified 1 - Left input, right output 2 - Right input, left output
rp_layout_fname	*string*	*(dp_name*). dpcm	Specifies the name of the resulting relative placement file. If you do not specify a name, the result is *design*.dpcm or *design*.tile.
rp_optimization	+|-	+	Turns optimization of column and row reordering, column merging, column removal, and alternative multiplier usage on or off. Second-level switch. (+|-)

Table 5-2 Relative Placement Environment Variables (Continued)

Module Compiler environment variable	Value	Default	Description
rp_pushpp	1/0	0	Specifies the resulting relative placement for an array multiplier, provided that maxtreedepth is set to 2 and multtype is set to nonbooth: 0 - Creates a parallelogram-shaped relative placement 1 - Creates a rectangular relative placement
rp_removal_rate	0.0 to 1.0	0.2	Sets the removal rate of a column based on percentage empty. By default, removes a column if it is 20 percent occupied or less.
rp_wallace	+|-	+	Sets the multiplier (+ to use alternative multiplier, – to use default multiplier). For best performance, set this value to minus (–).

Table 5-3 lists Module Compiler environment variables that are reserved.

Note:

Do not modify the following variables. Changing their values might cause Module Compiler to operate incorrectly.

Table 5-3 Reserved Environment Variables

Module Compiler environment variable
dp_buildlib_session1_name
dp_cd_ratio
dp_celldef
dp_cells
dp_cp_name
dp_cs_name
dp_datasheet_dir
dp_datasheet_fname
dp_dd_pp
dp_ddopt
dp_dir
dp_ds_name
dp_edf_name
dp_edif_lib_name
dp_flatlibname
dp_generic_lib
dp_generic_pf_lib

Table 5-3 Reserved Environment Variables (Continued)

Module Compiler environment variable
dp_genericlib_name
dp_help_dir
dp_include
dp_intermediate
dp_io_fname
dp_iterlibname
dp_lib
dp_libpath
dp_librep_out
dp_libs
dp_license_fname
dp_max_equiv
dp_max_inst_name_len
dp_pf_libs
dp_pseudo_lib_name
dp_release_libs
dp_repeat
dp_rio_name
dp_scan_fname
dp_scdf_lib_name
dp_scdf_name

Table 5-3 Reserved Environment Variables (Continued)

Module Compiler environment variable
dp_scdfdir
dp_show_err_window
dp_symbol_fname
dp_tbl_sort_asc
dp_tbl_sort_name
dp_tbl_sort_type
dp_tcl_name
dp_techdir
dp_techfile
dp_techindchk_fname
dp_ucp_name
tcl_dir
tk_dir

For information about UNIX environment variables and technology-specific Module Compiler environment variables, see “Installation and Setup” in Chapter 2 in the *Module Compiler User Guide*.

Command for Starting the Module Compiler GUI

Entering the following command at the command line loads the technology library you specify and then invokes the Module Compiler GUI:

```
mc -tech technology_library -gm +
```

For more information, see “Installation and Setup” in Chapter 2 in the *Module Compiler User Guide*.

dc_shell Commands for Running Module Compiler

Call the following commands from dc_shell to invoke Module Compiler and Module Compiler utilities and to run them in dc_shell-t mode (Tcl mode):

- `read_mcl` - Reads in a Module Compiler Language file
- `compile_mcl` - Synthesizes a Module Compiler Language design
- `mcenv` - Sets Module Compiler environment variables
- `report_mc` - Generates Module Compiler reports

These commands are described in “Overview of dc_shell” in Chapter 11 in the *Module Compiler User Guide*.

6

Error Messages

This chapter explains sample error messages that Module Compiler produces. It lists them in alphanumeric sequence, according to the references Module Compiler provides.

List of Error Messages

Module Compiler produces error messages at various stages during the processing and synthesis of input files. Where possible, Module Compiler provides a reference to the segment of the input that caused the error.

```
A1: Not authorized to run Module Compiler
```

No valid licenses could be found to run Module Compiler at any level.

```
A2: Module Compiler file (<string>) is corrupted
```

Module Compiler is running in a mode that permits synthesis only of authorized circuits. The input file either has been edited or it was generated by a tool that Module Compiler does not recognize.

```
A3: Not authorized to ....
```

A valid license was not found to allow the use of the stated advanced feature.

```
A4: Not authorized to ...
```

Access to these unreleased features is not allowed.

```
A5: Cannot find checksum entry for <name>
A6: Checksum mismatch for <name>
A7: Cannot open checksum file (<name>)
```

A checksum error has occurred, or the file is missing. Module Compiler is attempting to verify the library files. This problem must be corrected before Module Compiler can run.

```
F1: Could not open the file <string> for mode <string>
```

The file cannot be opened. Make sure you have permission to read from or write to it. Also be sure you have sufficient disk space if you are writing.

`G1: Spaces are not allowed in parameter values. parameter ='<string>', value ='<string>'`

A GUI parameter value contains a space. It must be removed to allow the values to be saved properly.

`I1: GUI main window init problem`

The GUI could not be started. Make sure the X Window System is running.

`I2: GUI tcl init problem`

The GUI code has problems. Check the Module Compiler installation.

`I3: GUI tk init problem...giving up`

The GUI code has problems. Check the Module Compiler installation.

`I4: GUI window init problem (<string>)... giving up`

The GUI code has problems. Check the Module Compiler installation.

`I5: CBA environment variable <string> undefined`

The specified cell-based array (CBA) environment variable is undefined. Check the CBA installation (make sure the UNIX environment variables—CBATECH, CBAVENDOR, and CBADIR—are set correctly).

`L1: Could not parse the SCDF file: <string>",name);`

The Module Compiler library or user library cannot be parsed. Check the installation for Module Compiler files, and make sure user files are correct.

```
L2: Cell <string> cannot be located in any library!
```

A cell could not be found in any library. This error typically occurs during reading of a netlist that contains cells that are not defined in the library.

```
L3: Cannot create instance of <string>. Was an EDIF file
read with the same name as the output?
```

Be careful not to create a module (cell) with the same name as any library cell.

```
L4: Unknown condition: <string> (should be 'fast', 'typ',
or 'slow')
```

Only the shown operating conditions are allowed.

```
L5: Unknown unateness parameter for '<string>'
```

Internal Module Compiler error. Contact the Synopsys Technical Support Center.

```
L7: I/O cell (<string>) not allowed
```

I/O cells are not supported. Remove these cells from the design.

```
L8: Could not read EDIF file: <string>
```

The EDIF file could not be parsed. This indicates some incompatibility between Module Compiler and the program that generated the EDIF. A more detailed error message is written to standard error (stderr).

```
R1: Pipelining failed at instance <string> (<integer>
<string>:<integer>)
```

Internal Module Compiler error indicating a pipelining problem. Contact the Synopsys Technical Support Center.

```
R2: Overloaded output '<string>' on instance '<string>'
load% 4.1f max: %4.1f
```

An instance has excessive load on the given output.

`R4: Overloaded input '<string>' load: %4.1f max: %4.1f`

An input of the module has excessive load (exceeds the maximum input load).

`R5: Overloaded net <string> cannot be fixed!`

The net specified has a constraint that makes fixing the overloading problem impossible. Make sure input max loads are not too small and output loads are not too large.

`R6: Pipelining in a loop detected, operand <string>[<integer] (behavioral model will not work) latency in: <integer> out: <integer>`

Pipelines were detected inside a loop, and the behavioral model does not work. Use latency hiding to prevent automatic pipelining.

`R7: This instance (<string>) cannot be optimized`

The overloaded net is driven by an instance that has logic optimization disabled and thus cannot be fixed.

`R8: Direct connection between input and output (<string>)`

The output indicated has a direct connection to an input of the module. This is not allowed in some netlists and frameworks.

`R9: Multiple tied outputs`

The EDIF file cannot be written when multiple outputs are connected to the same net. This is not allowed in some netlists and frameworks.

`OPT1:Could not run Design Compiler`

Make sure Design Compiler is installed correctly.

`OPT2:Design Compiler failed`

Design Compiler returned an error status. Check the error messages in the xterm in which Module Compiler was started, to determine the cause of the error.

```
SYN1:Attempt to pipeline CLK in operand <string>! Probably
deskewing inputs of a foreign synchronous cell
```

CLK cannot be pipelined, yet an instance with CLK as an input has been pipelined. This is not allowed, and latency hiding must be used to prevent automatic pipelining.

```
SYN2:Operand <string> has different delay constraint than
Group. Operand <string> (delay <integer>) is in Group
<string> (delay <integer>)
```

All operands within a single group must have the same delay constraint. This is required to enable equalization to work properly and for statistics to be reported properly.

```
SYN3:Buffering requested (<integer>) for <string> exceeds
max (<integer>)...setting to max value
```

The designer has used a buffer construct with a depth greater than that allowed. Reduce the depth.

```
SYN4:Delay goal <delay> is too small for pipelining, min is
<delay>
SYN4A:Adder Delay goal <delay> is too small for pipelining,
min is  <delay>
```

These warnings alert the designer to situations almost guaranteed to produce large latency and area. Either increase the delay goal or disable pipelining. The minimum delay goal is determined by the flip-flop output delay and setup time for each technology.

```
SYN5:Upper bit-range exceeds max for operand...adjusting
```

The MSB selected in the bit range is greater than that available for the operand. The real MSB is used instead.

```
SYN6:Lower bit-range is below min for operand...adjusting
```

The LSB selected in the bit range is less than that available for the operand. The real LSB is used instead.

```
SYN7:Could not build the full buffer tree
SYN8:Bit-range results in no bits selected...using 0
```

The bit range specified is entirely above or below the range of the operand. A 0 value is used instead. This condition should be corrected.

```
SYN9:Flatten failure <integer> (<string>)
```

The EDIF netlist could not be flattened. Check for illegal EDIF.

```
SYN10:Flatten failure, unresolvable loop detected in
<string>...timing might not be correct!
```

A continuous time loop was detected in an EDIF netlist file. The loop must be broken before Module Compiler can produce reasonable results.

```
SYN11:Input pin <string> is not connected
```

An input of an instance was never connected. All inputs must be connected.

```
SYN12:No RAMs could be built! Please check the parameters
```

No RAMs could be built. Common problems include illegal parameter values and an incorrectly installed RAM compiler.

```
SYN13:Could not open ram compiler output file <string>
```

A file permission or RAM compiler installation problem exists.

```
SYN14:Could not find the RAM cell <string>
```

Internal Module Compiler error in which the cell specified by the RAM compiler does not exist in any of the libraries produced. Contact the Synopsys Technical Support Center.

`SYN15:RAM created for another condition`

An attempt was made to use an existing RAM that was compiled for another operating condition or technology. Try deleting the old RAM SCDF files.

`SYN16:Too many inputs to MUX (max <integer>)`

Multiplexers can have only 1,024 inputs. Break larger MUXs into smaller ones.

`SYN17:Cannot convert <string>, no sign bit`

Not enough bits were computed to allow the creation of a sign bit (all constant signals). Try increasing the width of the sum construct into which this signal goes.

`SYN18: Cannot convert <string>, too many bits`

There were too many bits (>2) in one or more bit positions. Make sure `convert` was used as an option to `carrysave`.

`SYN19:Latency hiding failed at <string>`

An attempt was made to hide latency in which the reference has greater delay than the signal being adjusted. This situation requires negative delays and cannot be handled.

`SYN20:Equalization failed at <string> (need length <integer>)`

An attempt was made to equalize the delays in which the reference has less delay than the signal being equalized. This situation requires negative delays and cannot be handled. The indicated value is the difference between the latency of the reference and the latency of the input.

SYN21:Cannot pipeline a foreign cell (<string>)...turning

off

To avoid catastrophic problems, pipelining is allowed only for combinational cells Module Compiler knows.

SYN22:Cannot pipeline a RAM...turning off

To avoid catastrophic problems, pipelining is allowed only for combinational cells Module Compiler knows.

SYN23:Bus operand (<string>) has too many bits (<integer>) in column <integer>

Simple concatenation cannot be used to form the operand. Remove signals from the bit position indicated.

SYN24:Bus operand (<string>) has 0 bits in column <integer> (value will be 0)

The resultant operand has “holes” that will be filled with 0.

SYN25:Ignoring <string> outside of carrysave

Only `carrysave` can use `convert` and `optimize`.

SYN26:Ignoring final adder type in carrysave

The `carrysave` operands need no final adder.

SYN27:Attempt to perform more than one rounding operation ... ignoring

It is not meaningful to round a result more than once.

SYN28:Cannot round to bit pos 0 ... ignoring

It is not meaningful to round a result to the LSB.

SYN29:Only using MSB of offset <string>

The offset input of a Booth multiplier can use only a single bit. The MSB is used.

SYN30:Killing nontransient operand (<string>)!

Internal Module Compiler error. Contact the Synopsys Technical Support Center.

SYN31:Select value (<integer>) is out of range for this MUX (<integer>)

A select value was declared for a MUX that is outside the range supported by the select input for the MUX.

SYN31A:Select input (<string>) is too wide in 'MUX'. Max width is 12

Module Compiler supports only select inputs with up to 12 bits. Reduce the width of the select input.

SYN32:LSB for input signals must have index 0

All operands must have bit ranges starting at 0.

SYN33:All inputs must be in the first cycle

All inputs must have arrival times that are less than the delay goal (cycle time).

SYN34: (Fatal) Operand for instance <string> has no permanent version

Internal Module Compiler error. Contact the Synopsys Technical Support Center.

SYN36:FB Reg input operand bit width mismatch for <string>

The feedback shift register references an operand with a different width than the width specified. Correct one of the widths.

SYN37:Converting signed operand (<string>) to unsigned for this operation

A signed operand was supplied to an operator that supports only unsigned operands. The sign bit is treated as unsigned.

SYN38:Unknown scan ff conversion <string>

Internal Module Compiler error. Contact the Synopsys Technical Support Center.

`SYN39:Scan FF scan pin on <string> has been used, not scan compatible`

In conversion of the scan flip-flops back to D flip-flops, one was found with a scan input in use. The instance cannot be converted, and the design cannot support scan testing.

`SYN40:FF instance <string> is not scan compatible`

The flip-flop instance has no equivalent scan test flip-flop that is known to Module Compiler. Remove it, or disable scan test mode.

`SYN42:Instance name <string> is too long`

The name is too long for SPICE netlisting. Shorten the group name.

`SYN43:Select input is not wide enough for demux (<string>)`

The select input does not have enough bits to cycle through 0 to n–1, as required. Increase the width of the select input.

`SYN44:It is not meaningful to have a constant select input for demux (<string>)`

The demultiplexer requires an active select input to operate. If the select input is constant, the demultiplexer outputs the input delayed by two cycles or it is always undefined.

`SYN45:Nonoperand (<string>) specified in 'parameters' construct`

The operand in the parameter list is not an operand in the module. Remove the name.

SYN46:Operand (<string>) specified in 'parameters' construct

`is not an I/O`

The operand in the parameter list is not an input or output of the module. Only inputs and outputs are allowed in this construct.

`SYN47:Cannot run the preprocessor <string>`

The language preprocessor could not be run. This indicates a UNIX or CBA setup problem.

`SYN48:Cannot run the RAM compiler`

The RAM compiler could not be run. This indicates a UNIX or CBA setup problem.

`SYN49:Shift/Rotate require input or output to have 0 LSB index at <string>`

The synthesis routines will not work unless this condition has been met.

`SYN50:Ignoring 'optimize' in pipelined operand (<string>)`

An attempt was made to use the `optimize carrysave` option with pipelining enabled. This does not work well, and the `optimize` is ignored.

`SYN52:Delay must be greater than zero, got <value>`

Negative delay goal values have no meaning. They must be positive.

`SYN55:Design has too many ports to generate database.  Max is <max>`

The synthesized cell has too many ports (each bit of a bus is counted as 1 port). The limit is very large (~32,768).

`SYN56:Cannot find the clock operand <name>`

The operand specified in the `clockbuf` statement does not exist.

```
SYN57:Cannot find the clock net <name>
```

The clock operand could be found, but the net could not. This can occur if a dummy operand that does not have any real nets is referenced.

```
SYN59:CLK net doesn't have 1 pin, buffering failed
```

An internal Module Compiler error has occurred, and the clock buffering failed. Contact the Synopsys Technical Support Center.

```
SYN60:Input pin <name> is connected to more than one input
SYN61:Input pin <name> is unconnected
SYN62:Input pin <name> does not drive an I/O buffer
SYN66:Input pin <name> does not connect to a 'pad' input of
the I/O buffer
```

Top-level input pins must be connected to only a single input buffer pad.

```
SYN63:Output pin <name> is not driven by an I/O buffer
SYN67:Output pin <name> does not connect to a 'pad' input
of the I/O buffer
```

Top-level output pins must be connected to the “pad” output of an output buffer pad.

```
SYN64:I/O buffer <name> is not connected to an I/O pin
```

All I/O cells must be connected to a top-level I/O pin.

```
SYN65:I/O buffer <name> found in a design which is not
top-level
```

I/O buffer cells are allowed only in a top-level design.

```
SYN66:Input/Bidirect pin <name> does not connect to a 'pad'
input of the I/O buffer
```

In a top-level design, inputs and bidirects of the module must connect to a pad input of the I/O buffer.

```
SYN67:Output pin <name> does not connect to a 'pad' output
```

```
of the I/O buffer
```

In a top-level design, outputs of the module must connect to a pad output of the I/O buffer.

```
SYN68:Improper loop at <name>. Timing will be incorrect
```

A continuous time loop was created, or the fb-type input was used incorrectly. Module Compiler can continue, but the timing in this part of the circuit will not be accurate.

```
SYN69:Concat operator <name> has too many bits <num> in
column <num>
```

The internal concatenate construct is malformed.

```
SYN70:Concat operator <name> has no bits in column <num>
```

The internal concatenate construct is malformed.

```
SYN71:Critical path input|output <name> is not
defined...ignoring
```

The operand specified as the path startpoint or endpoint could not be found. No analysis is performed.

```
SYN72:'Latch' expects a 1-bit enable input
```

The internal latch construct is malformed.

```
SYN73:Illegal critical path type <type>
```

The internal critpath construct is malformed.

```
SYN74:Cannot connect an input <name> to a tristate net
```

A module input cannot be connected to a three-state net. It must be buffered inside the module.

```
SYN75:Bidirect pin <name> does not connect to a bidirect
I/O buffer
```

All module bidirects must connect directly to a bidirect I/O buffer.

SYN76:Input pin <name> does not connect to an input I/O buffer

All module inputs must connect directly to an input I/O buffer.

SYN77:Must resynthesize design with CLK buffer or scan mode

These two modes change the network during report generation. The circuit must be resynthesized before continuing.

SYN78:Net <name> with latency <num> from clock <name> entering group with clock <name>

Latency deskewing is not allowed between clocks. Use `hidelat` to remove latency from signals from one clock before interaction with signals from another clock.

SYN79:Group <name> has two clocks (old=<name>, new=<name>)

Each group is allowed to have only one clock. Two were detected in the group.

SYN80:Width for demux <name> is too small or not specified (2 is min)

Demultiplexers must have at least two outputs.

SYN81:Bused instance of <name> has too many inputs (> <num>)

Bused instances can have only 256 inputs.

SYN82:Requested clock driver <name> is an input...not connecting

The clock net must be driven from an instance when buffering is requested. It is not meaningful to supply an input.

SYN83:Clock (<name>) was previously declared

A signal declared via the `clock` attribute was declared as a wire, input, or output. This is not allowed. Module Compiler automatically creates a global signal that is a module input.

IS1: Bad int

An integer was expected.

IS2: <iomode> in <string> on cell <cell> cannot be found

The specified pin could not be located on the cell.

IS3: Attempt to redefine <string>

An operand with the same name has already been defined. Choose another name.

IS4: Syntax error: could not find ']' in bit-range

A bad bit range syntax was encountered.

IS5: Operand, <string>, has not been defined

An operand has been referenced that has not been defined yet. Make sure it is defined earlier in the file.

IS6: Not allowed to select a range of carrysave operand <string>

`carrysave` does not require or allow bit ranging operations.

IS7: Negative shift value is not allowed at <string>, please use bit-range instead

A fixed negative shift is equivalent to the truncation provided through bit ranging. Bit ranging is preferred.

IS8: Bit ordering backward...corrected

The MSB index must come first.

IS9: Output range has no meaning for <string> (use input range instead)

The range for the output is not used. The range is chosen automatically from the width of the input operand.

`IS10:Expected ')'`

Syntax error.

`IS11:Expected string`

Syntax error.

`IS12:Expected '('`

Syntax error.

`IS13:Missing keyword`

Syntax error.

`IS14:Unexpected EOF in 'rem'`

Comment contains the EOF. This is probably not intended.

`IS16:Bit-range not allowed for pin name`

Bused pins of an instantiated cell cannot contain bit ranges. The entire bus must be connected in a single statement.

`IS17:Duplicate connections to <string>`

An attempt was made to connect to a pin with more than one net.

`IS18:Bad keyword in function <string>`

Syntax error.

`IS18:Bad keyword in function <string>, subfunction <string>`

Syntax error.

`IS19:Expected parameters`

RAM compiler parameters must come before inputs and outputs.

`IS20:Operand (<string>) with carrysave format is not allowed here ...`

This function does not support carry-save operands.

IS21:This type of MUX cannot be inverted for <string>

The AND-OR and TRISTATE MUXs don't support output inversion.

IS22:Offset is allowed only in Booth Multipliers (amult and bmult)

Non-Booth multipliers have no offset input.

IS23:Reference operand <string> is not defined

An attempt was made to equalize delay or hide latency by use of an undefined reference. Make sure the references are defined earlier in the file.

IS24:'Convert' is only for carrysave operands
IS25:Feedback Shift Register (<string>) cannot have a 0 length

To prevent the creation of continuous time loops, zero-length feedback shift registers cannot be created.

IS26:Cannot normalize right

Normalize always works in the left direction. Flip the bits to normalize right.

IS27:Shift operand must be specified in 'shift', 'rotate', or 'normalize' No shift operand was specified. One is required.

Choose one of the legal values.

IS28:Illegal primary optimization scheme <string> (should be 'speed', 'power', 'size', or delay value)

Choose one of the legal values.

IS29:Illegal secondary optimization scheme <string> (should be 'power', or 'size')

Choose one of the legal values.

IS30:Unknown logic opt mode <string> (should be 'on' or 'off')

Choose one of the legal values.

IS31:Unknown pipeline mode <string> (should be 'on' or 'off')

Choose one of the legal values.

IS32:Expected 'cell'

Syntax error. The cell definition must come first.

IS33:EOF expected (possible extra ')')

Some characters are ignored at the end of the file, because they are usually unintended.

IS34:Illegal symbol type <string> (should be 'tsg' or 'pinlist')

Choose one of the correct symbol types.

IS35:Reusing a select value in this MUX

There are two inputs using the same select value. This is not allowed.

IS36:Null integer supplied

No value was supplied when expected.

IS37:Expected <integer> inputs at <string>, found <integer>

An incorrect number of inputs was supplied to the function. Some functions require a particular number of inputs.

7

Performance Data

This chapter shows comparisons between the structures and architectures available in Module Compiler, in the following sections:

- 2-Input Adders
- Incrementors
- Comparators
- Multipliers
- Decoders
- Multiplexers
- Shifters and Rotators
- Normalized
- Equality Compare

- AND, OR, XOR Logic
- Symmetric Pipelined FIR Filters
- Asymmetric Pipelined FIR Filters

The performance data in the following sections is generated with an estimated wire loading, 0.8-µ CMOS process at WCCOM operating condition (1.86 factor relative to typical). All timing values have units of ns. Although this is an older technology, the trends are still relevant.

The following graphs are in color online and are easiest to view there. To view these graphs online, see Chapter 7, “Performance Data,” on Synopsys Online Documentation (SOLD).

The area is the total number of sections occupied before routing. The compute-to-drive ratio, which can be extreme, is not taken into account. Some optimization is performed, but for the sake of time, the amount of optimization is kept to a minimum. Some improvement in this data would likely be obtained with further optimization. All input signals are assumed to arrive without skew, unless the opposite is explicitly stated.

2-Input Adders

The data in Figure 7-1 shows the raw performance of the four final adder architectures for a given width, n. Each was optimized for speed. The clsa and the fastcla provide nearly identical performance under these conditions. It can be seen that the cla, the clsa, and the fastcla all provide logarithmic delay whereas the csa provides delay proportional to $\sqrt{n}$ and hence performs poorly for large n.

Figure 7-1 Adder Delay Versus Width

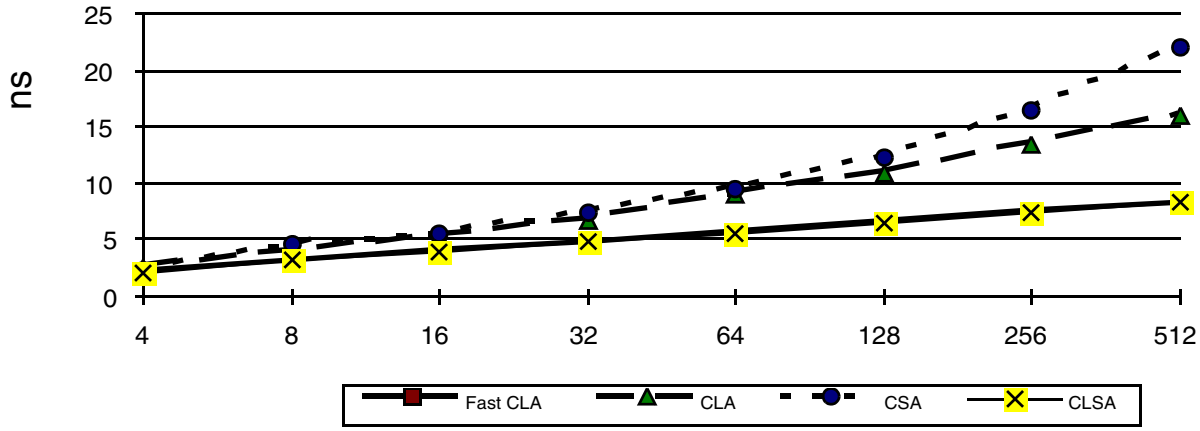
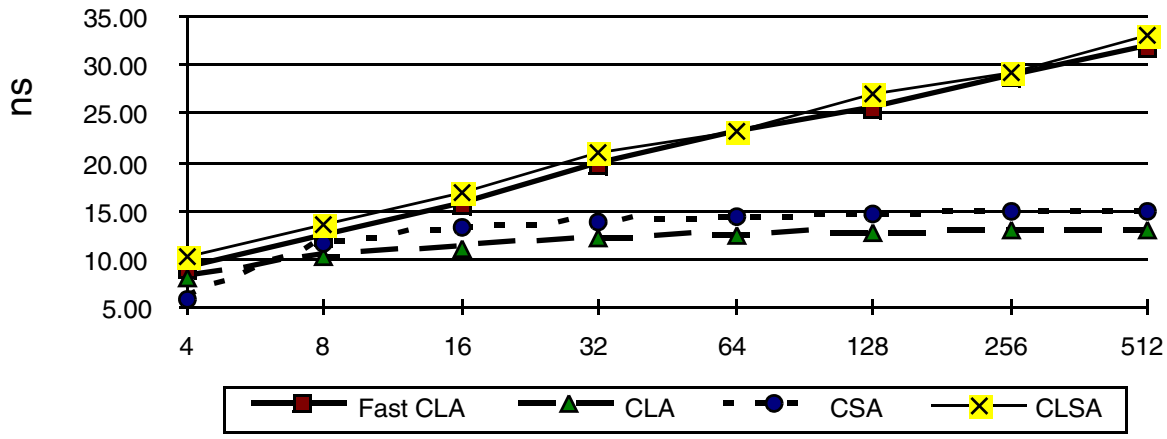


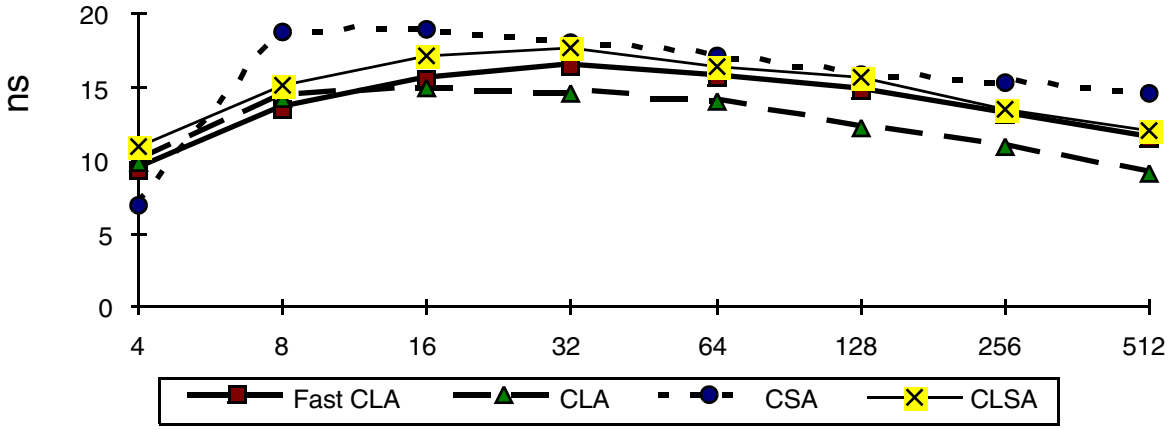
Figure 7-2 shows the area per bit. Note that the clsa and the fastcla provide an area per bit proportional to log 2(n) whereas the csa and the cla have a constant area per bit for large n.

Figure 7-2 Adder Area per Bit Versus Width



To compare the adders, it is useful to consider the normalized (by n1.5) area-delay product (see Figure 7-3). The measure should be roughly constant for carry-select adders (csa) when measured against the adder width; the area is proportional to n, and ideally, the delay is proportional to $\sqrt{n}$. By this measure, the cla generally provides the best performance, because the csa is too slow and the fastcla and the clsa use too much area for a given performance level.

*Figure 7-3 Adder Area * Delay Versus Width, Normalized*



Consider a 64-bit 2-input adder that is optimized for a desired delay rather than for speed. Figure 7-4 shows the actual versus desired delay for this case. The clsa can accommodate a greater range of desired delays than the csa. The fastcla and the cla have very little ability to adapt to changing desired delays.

Figure 7-4 Adder Delay Versus Desired Delay

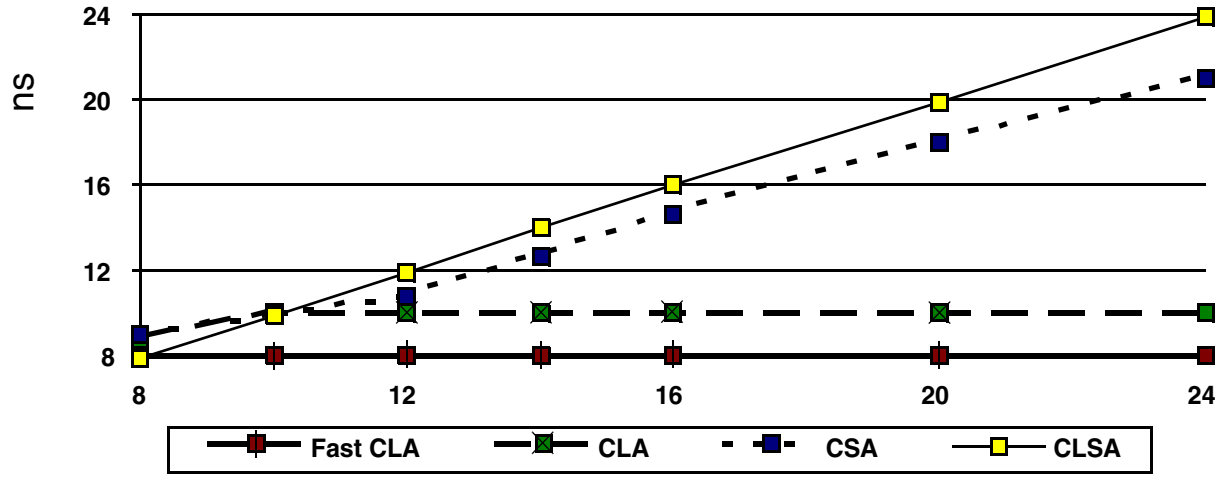
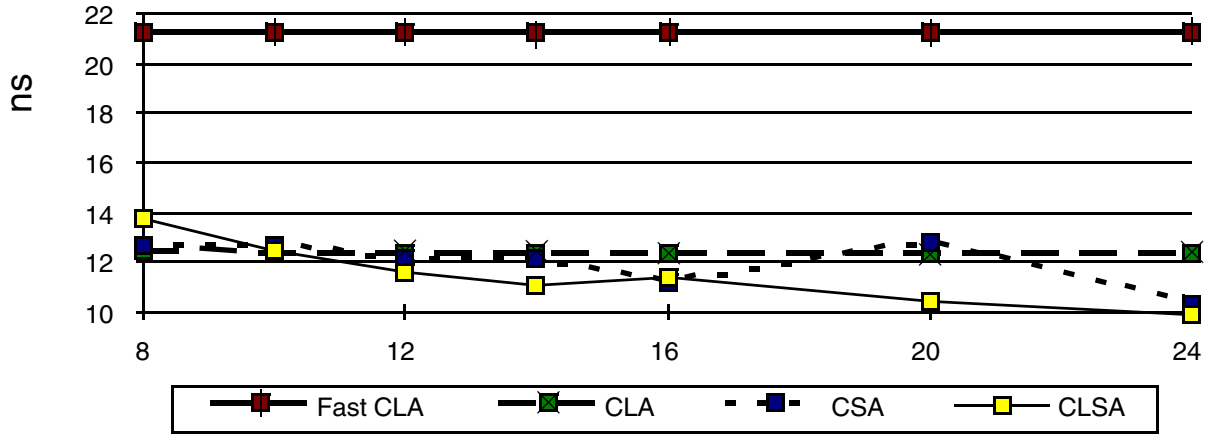


Figure 7-5 shows the area per bit. Note that the clsa shows dramatic gains as the desired delay is first increased but that diminishing improvement follows. The csa shows only slight area improvement as the desired delay increases.

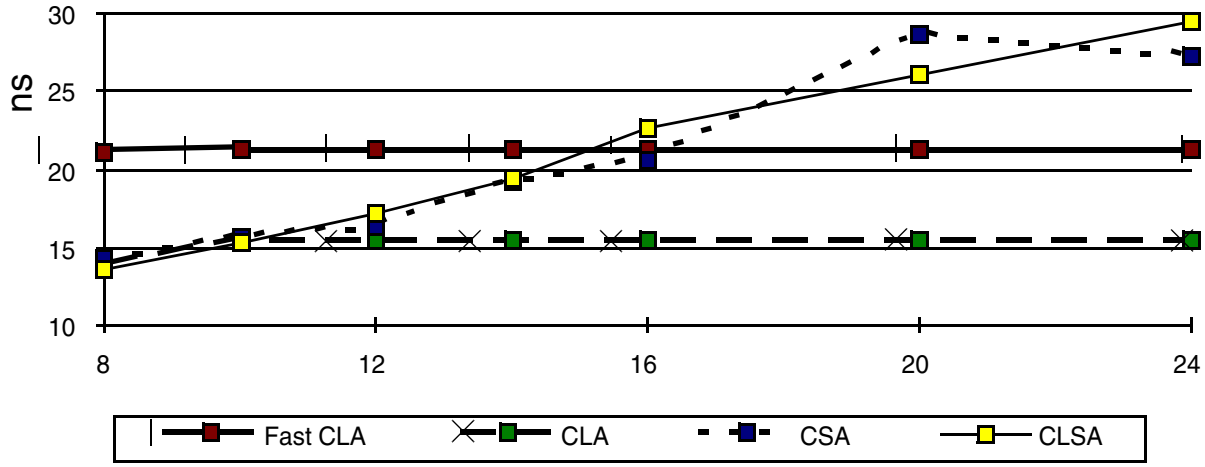
Both structures degenerate into ripple adders as the desired delay approaches infinity. It is interesting that the fastcla and the cla show virtually no change in area as the delay goal is increased, whereas the actual delay does increase. Although this might appear to be giving up delay for nothing, the optimizer does decrease the area by making the compute-to-drive ratio closer to 3.0. However, this effect does not show up directly in the results.

Figure 7-5 Adder Area per Bit Versus Desired Delay



The plot in Figure 7-6 of the normalized area-delay product shows the efficiency of increasing the desired delay. The clsa clearly is more efficient when not optimized for speed but quickly becomes more inefficient as the desired delay is increased further.

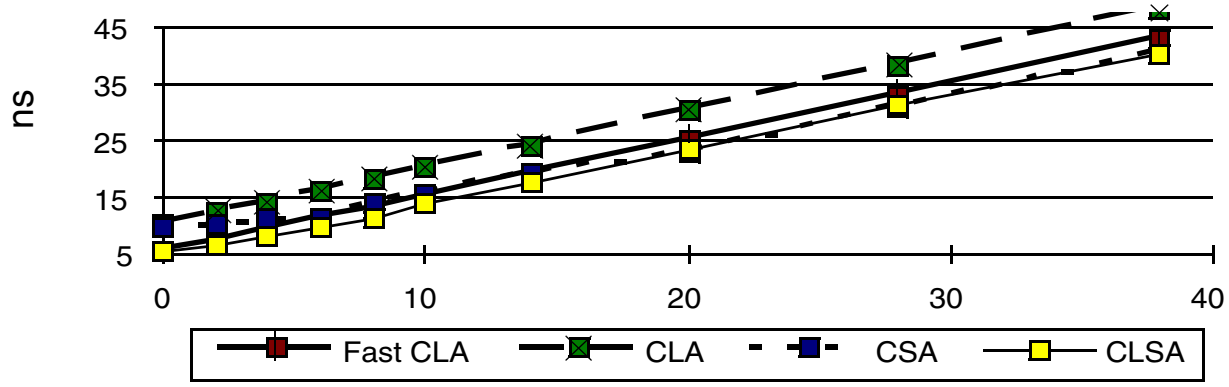
Figure 7-6 Adder Area-Delay Product Versus Desired Delay



Another interesting case to examine is when the inputs do not have uniform arrival times. Consider the 64-bit adder with an additional 1-bit input in the LSB that arrives at some time after the other inputs. The plot in Figure 7-7 shows the critical path delays for each adder architecture as the delay skew is changed.

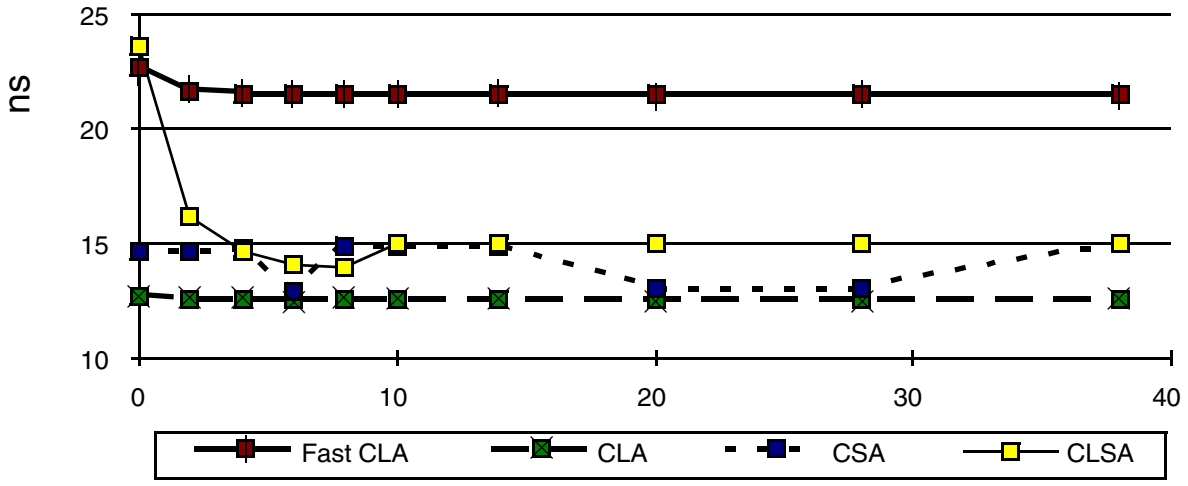
Note that the csa and the clsa can change their structures to reduce the critical path delay as the skew increases. In fact, for large skews, the csa provides better performance than the fastcla. The clsa provides the best delay performance overall.

Figure 7-7 Adder Delay Versus Slow Input Delay



The plot in Figure 7-8 of the area per bit versus the input delay skew shows that the clsa provides area improvements in addition to the delay improvements. The csa shows slight area degradation; this is caused by the shortening of the first (ripple) stage, which can be constructed more efficiently than the carry-select stages.

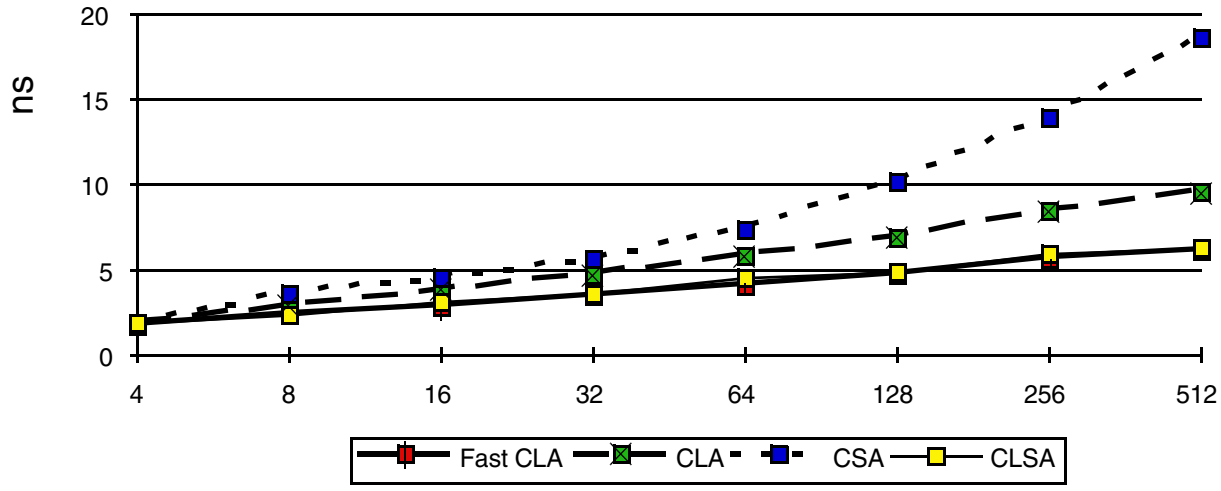
Figure 7-8 Adder Area per Bit Versus Slow Input Delay



Incrementors

Figure 7-9 shows how well the previous adder structures degenerate into incrementors. As expected, the delay characteristics are similar to but, in general, better than those obtained for the adder case.

Figure 7-9 Incrementor Delay Versus Width



Also, the area per bit for incrementors follows a similar pattern to that of the adders, with the primary exception being that the csa and the cla provide similar area (see Figure 7-10 and Figure 7-11).

Figure 7-10 Incrementor Area per Bit Versus Width

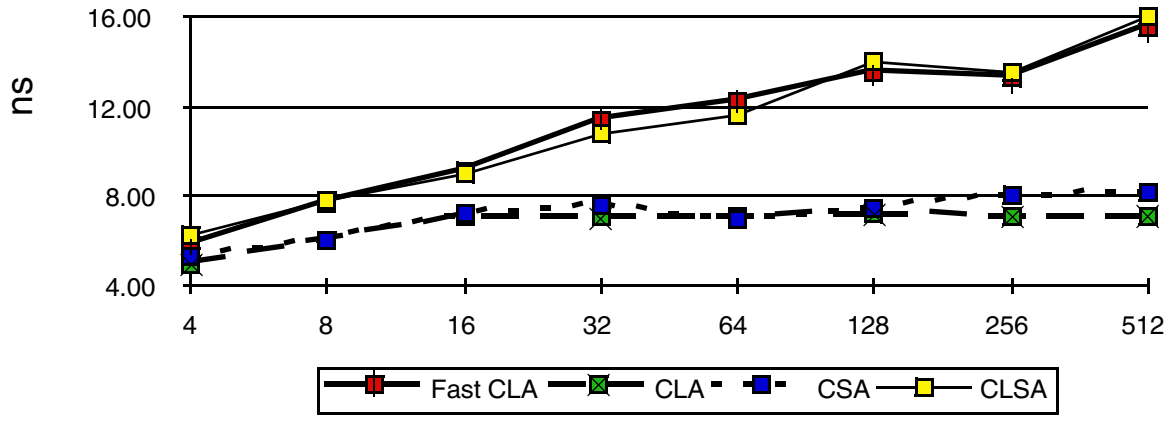
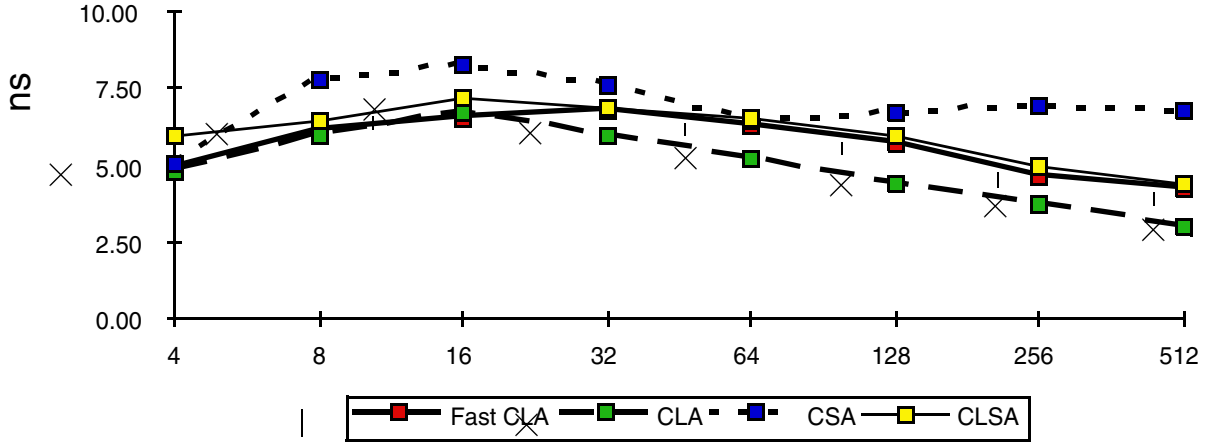


Figure 7-11 Incrementor Area-Delay Product Versus Width, Normalized



Comparators

The comparator is another degenerate case of the 2-input adder. The most interesting part of this case is that the cla and the fastcla degenerate into the same structure, resulting in similar area and delay (see Figure 7-12).

When used as a comparator, the inverted carry tree (the second part) of the cla is removed, improving the delay (see Figure 7-13). The majority of the carry tree of the fastcla can also be removed, improving area. In addition, the carry-lookahead structures have lower area and delay than the csa.

The csa, which is clearly not a good choice for comparators, does not always degenerate properly if the delay constraint is set too small. Because of the similarity of results (except for the csa), Module Compiler always uses the fastcla for comparators (see Figure 7-14).

Figure 7-12 Comparator Versus Delay Width

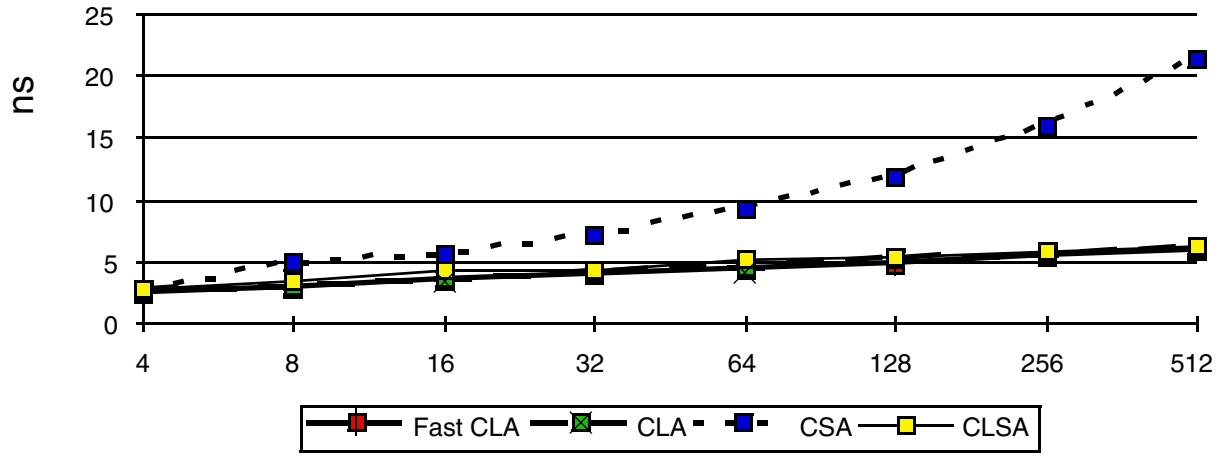


Figure 7-13 Comparator Area per Bit Versus Width

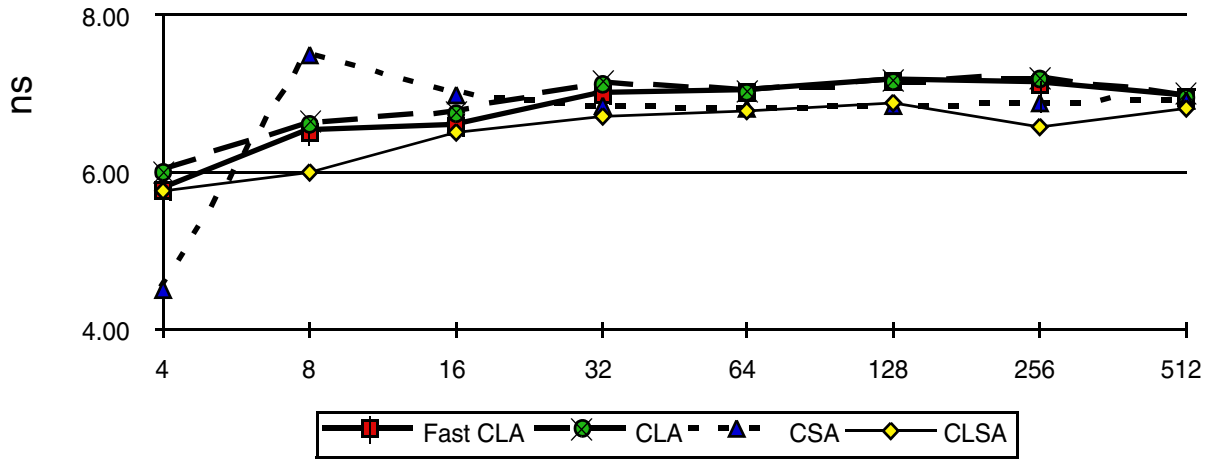
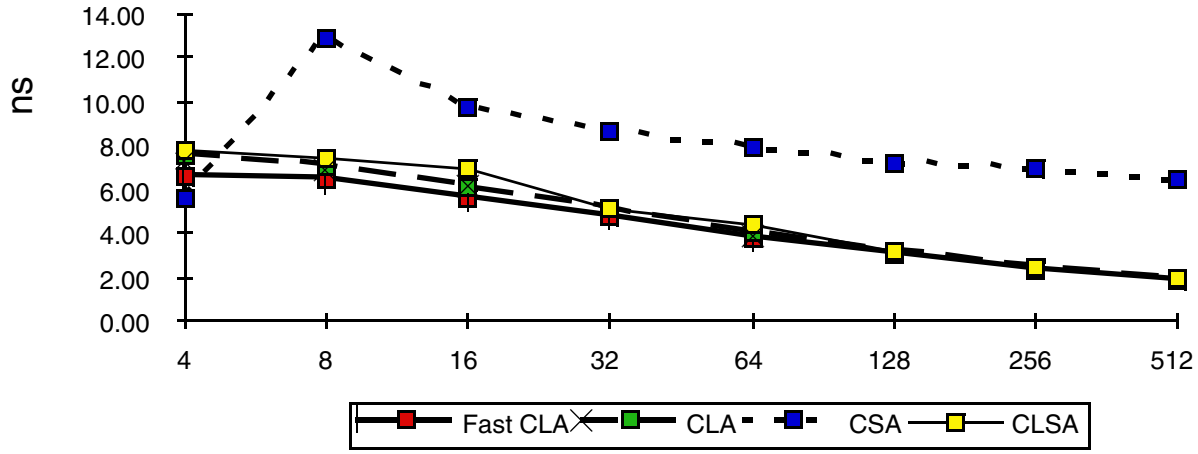


Figure 7-14 Comparator Area-Delay Product, Normalized



Multipliers

Multipliers are more complex to characterize than adder structures, because the multiplier and final adder architectures and the size and formats of the two inputs all affect the results.

In all cases examined, both inputs have the same width and are signed. The tradeoffs between the Booth and non-Booth multipliers are affected by this assumption.

First, consider the effects of the multiplier architecture with all other variables fixed (see Figure 7-15 and Figure 7-16). The Booth multiplier has smaller delay and area when the inputs have widths greater than 16 and 7 bits, respectively. A good rule of thumb is to choose Booth-encoded multipliers for 8 x 8 and above.

Figure 7-15 Multiplier Delay Versus Width

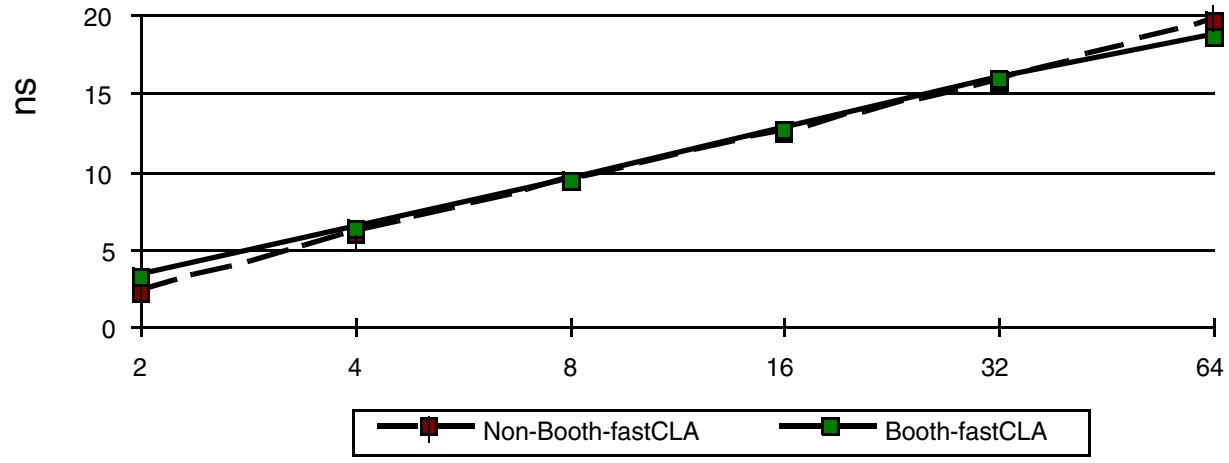
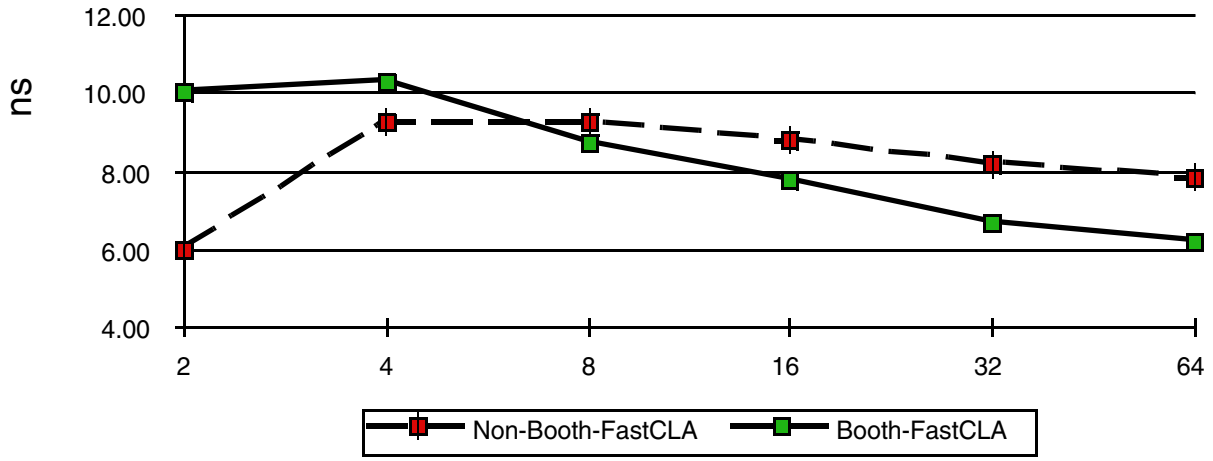


*Figure 7-16 Multiplier Area per Bit*Bit Versus Width*



The final adder architecture also affects the performance of the multiplier. Because the delays at the output of the Wallace tree are highly skewed, the csa and the clsa provide area advantages. However, the delay skews are not large enough to keep the fastcla from providing the best performance. Figure 7-17, Figure 7-18, and Figure 7-19 show that the csa and clsa final adder architectures are best in terms of overall efficiency.

Figure 7-17 Booth Multiplier Delay Versus Width

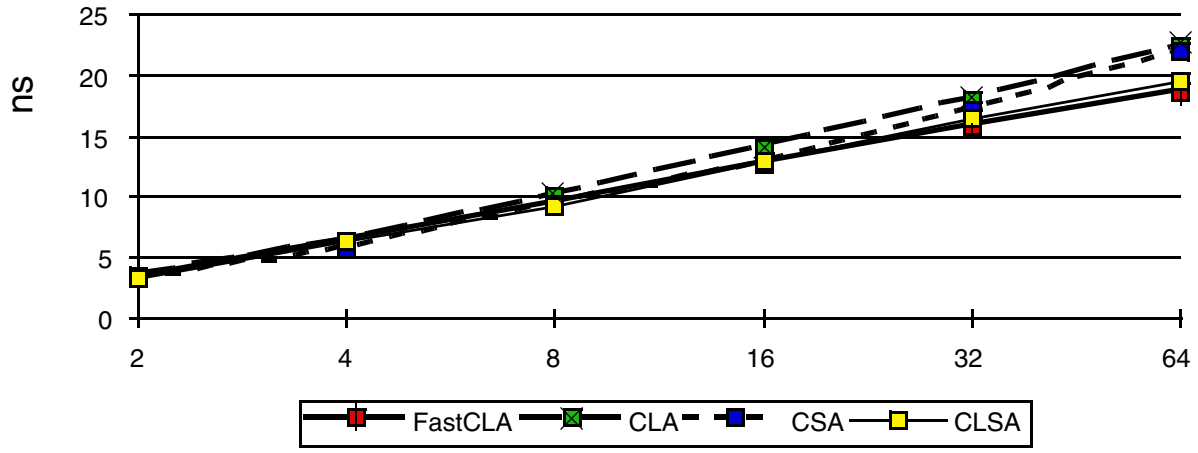


*Figure 7-18 Booth Multiplier Area per Bit*Bit Versus Width*

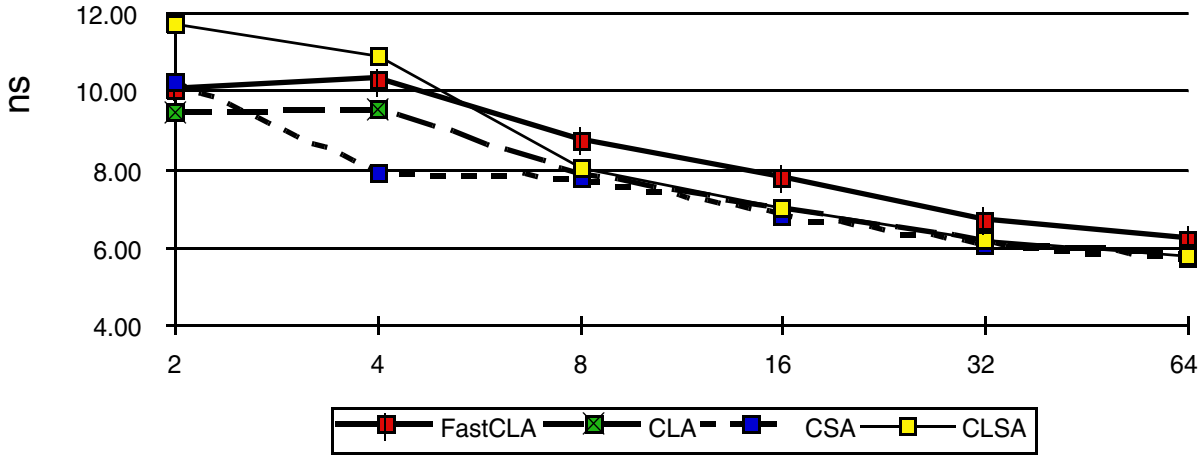
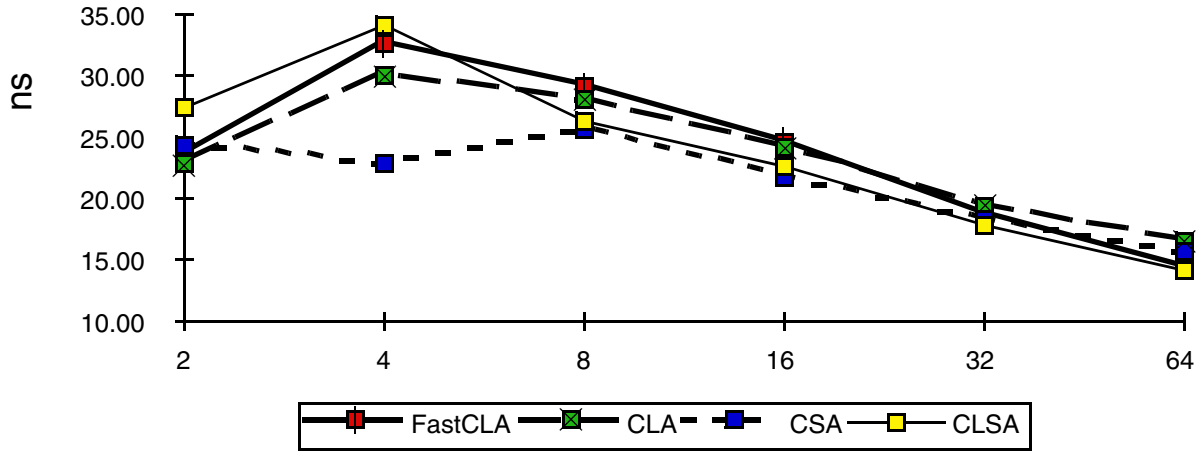


Figure 7-19 Multiplier Area-Delay Versus Width



Decoders

Figure 7-20 and Figure 7-21 show the relationship between the number of input bits and the delay and area for various decoders. Each decoder was optimized for speed.

Note that the area has been normalized by m * n –1, where m is the number of outputs (m = 2n) and n is the number of input bits. This should provide a rough measure of area in terms of 2-input AND gates.

Interestingly, for large n, a value very close to 1 is achieved, as expected. For small values of n, the inverters and buffers used to drive the AND gates become an appreciable fraction of the total area.

Figure 7-20 Decoder Area per M n –1 Versus n*

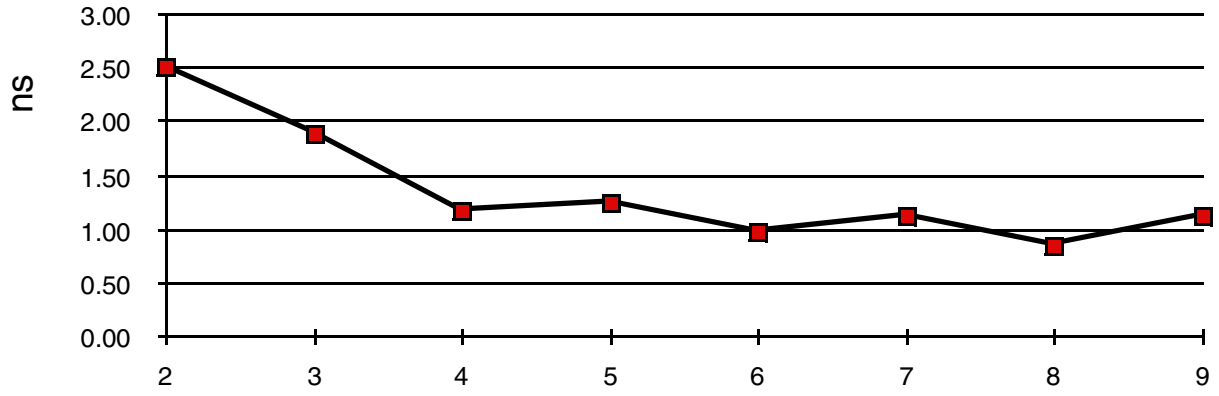
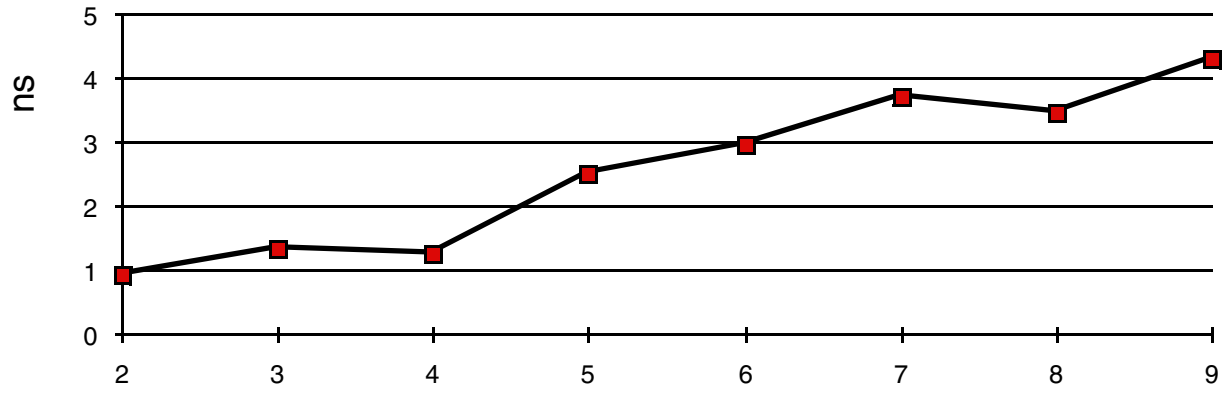


Figure 7-21 Decoder Delay Versus n



Multiplexers

Figure 7-22 shows the relationship of delay to the number of inputs. Note that the ANDOR and MUX-based structures provide generally logarithmic delay.

The ANDOR structure incurs the additional cost of the decoder, making it inferior for the no-skew case.

The three-state structure has a flatter delay characteristic at first, but the delay rises rapidly as the outputs of the driver become overloaded (no optimization is performed on the three-state drivers).

For all cases, n is the number of inputs.

Figure 7-22 MUX Delay Versus n (No Skew)

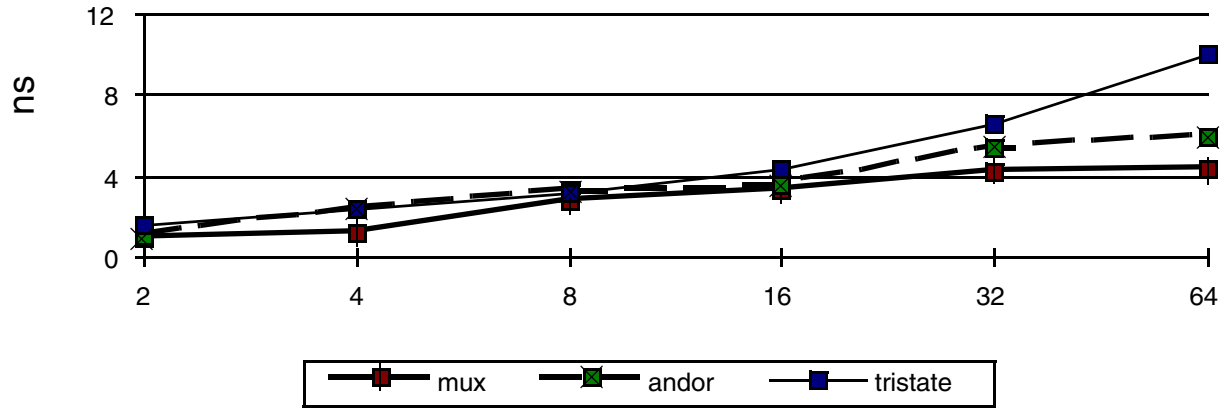
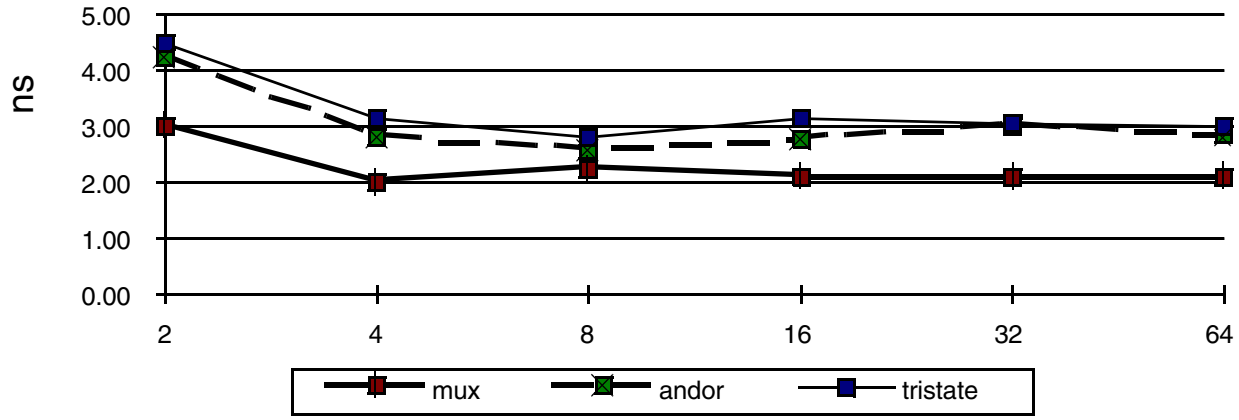


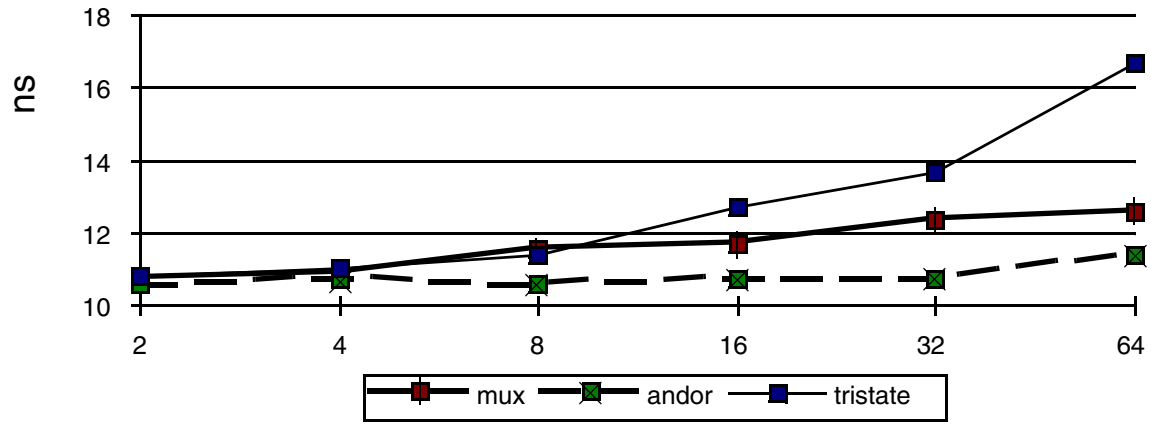
Figure 7-23 shows that the MUX-based structure is also significantly more area-efficient than the others.

Figure 7-23 MUX Area per Bit Versus n, Normalized to n –1 (No Skew)



When the data inputs are skewed, the ANDOR structure can be used to provide some delay improvement. Figure 7-24 shows a case in which one input arrives 10 ns after all the others.

Figure 7-24 MUX Delay Versus n (10-ns Skew)



Shifters and Rotators

To characterize the shift and rotate functions, consider cases (see Figure 7-25, and Figure 7-26) in which the data inputs and outputs have the same width and the depth is chosen to allow all possible shifts—that is, depth = $\log_2$(width).

These functions also generally provide logarithmic delay versus width.

Figure 7-25 Shift, Rotate Delay Versus Width

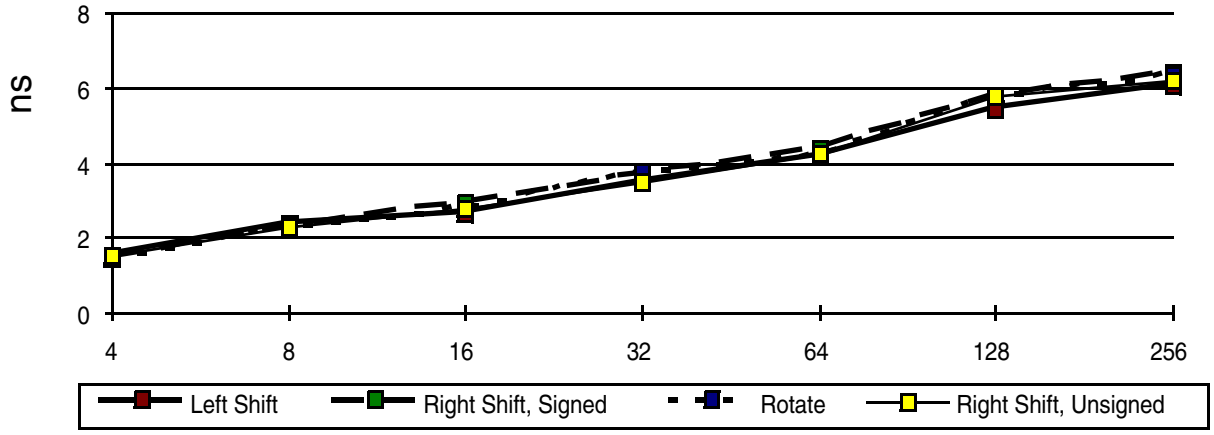
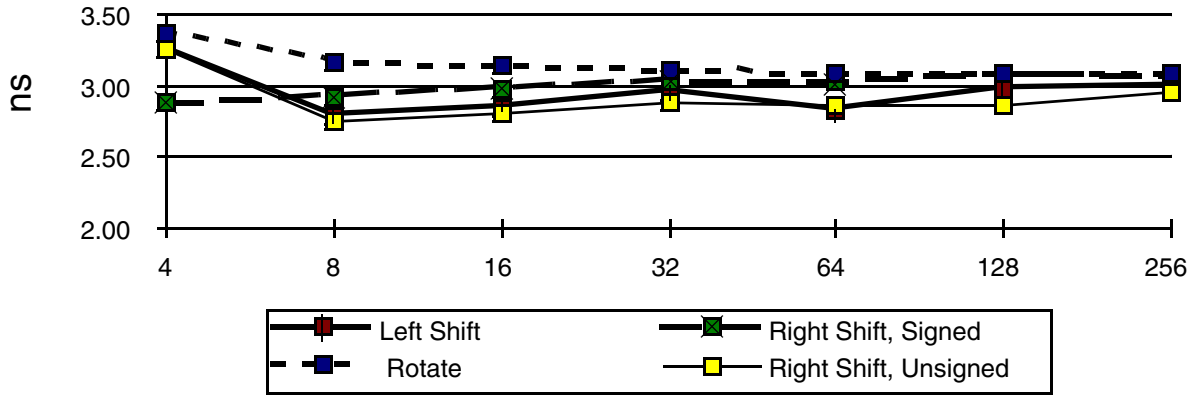


Figure 7-26 Shift, Rotate Area Versus Width, Normalized by Width and Depth



Normalized

Figure 7-27 and Figure 7-28 show normalized delay versus width and normalized area versus width.

Figure 7-27 Normalized Delay Versus Width

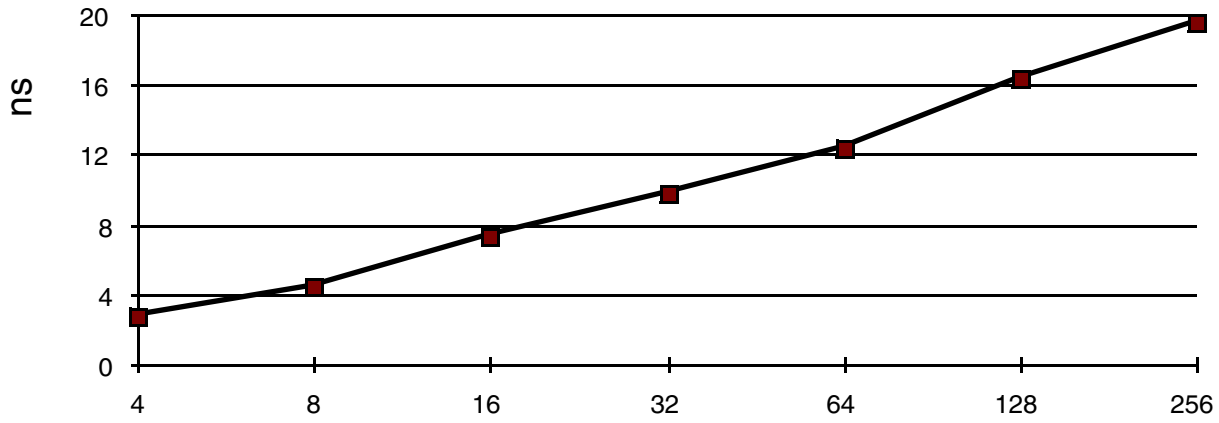
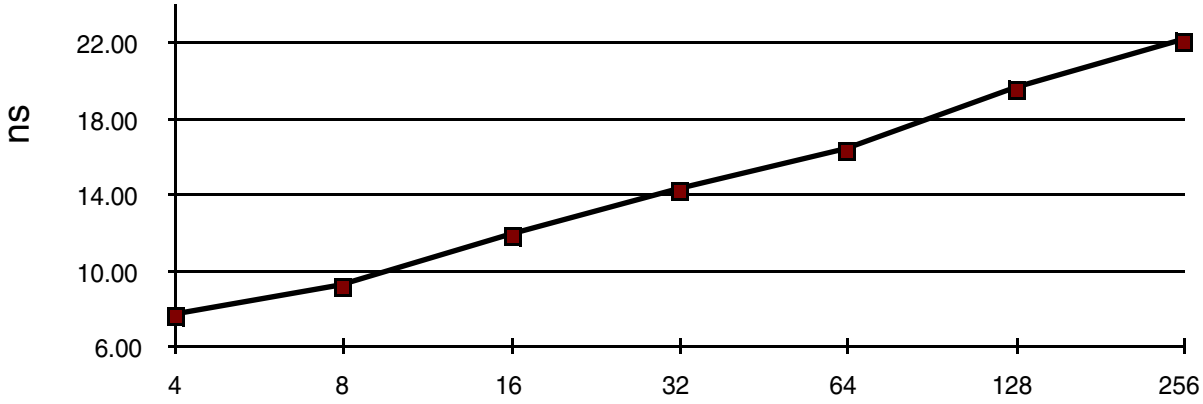


Figure 7-28 Normalized Area per Bit Versus Width



Equality Compare

Figure 7-29 and Figure 7-30 show equality delay versus width and equality area versus width.

Figure 7-29 Equality Compare Delay Versus Width

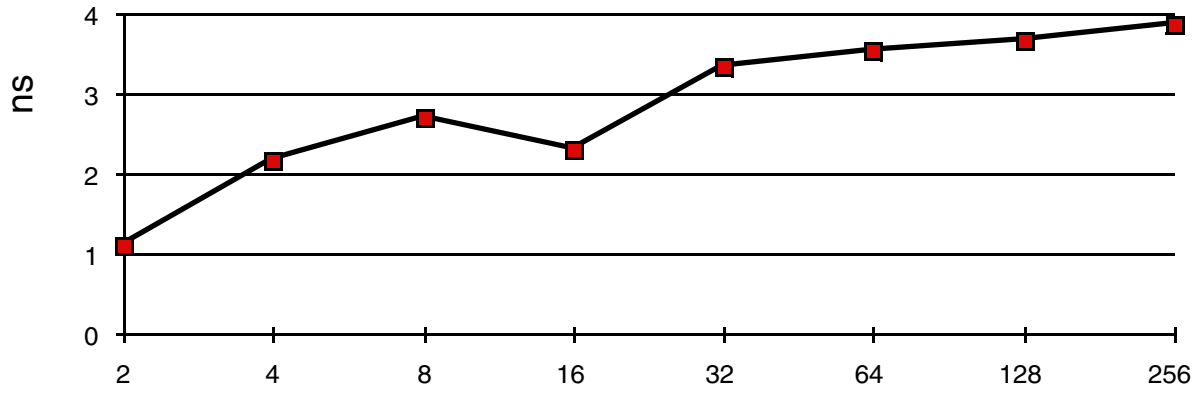
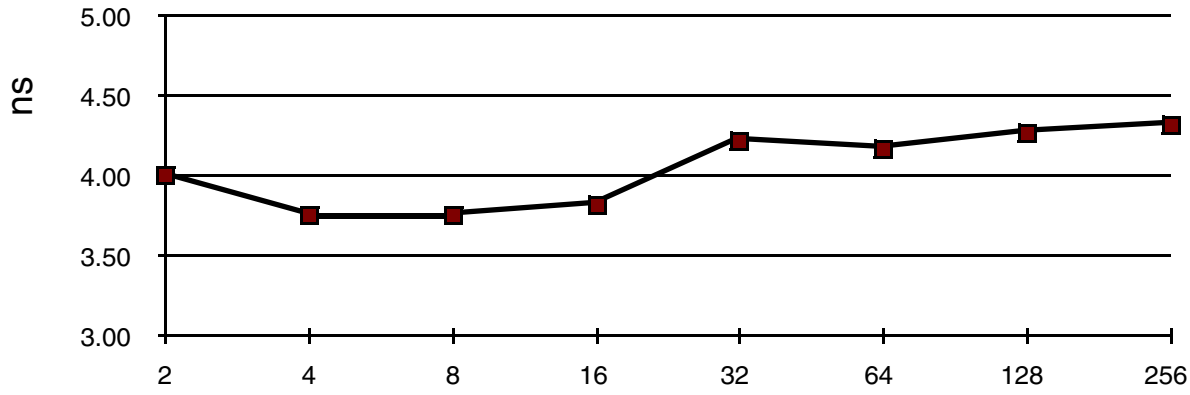


Figure 7-30 Equality Compare Area per Bit Versus Width



AND, OR, XOR Logic

Consider the performance of large logic trees (see Figure 7-31 and Figure 7-32). Note that in all cases, generally logarithmic delay performance is obtained. AND and OR logic optimized for speed (fast-AND and fast-OR) results in generally smaller delays (except for 16 inputs). As expected, the size-optimized trees provide better area.

Figure 7-31 AND, OR, and XOR Delay Versus Tree Depth

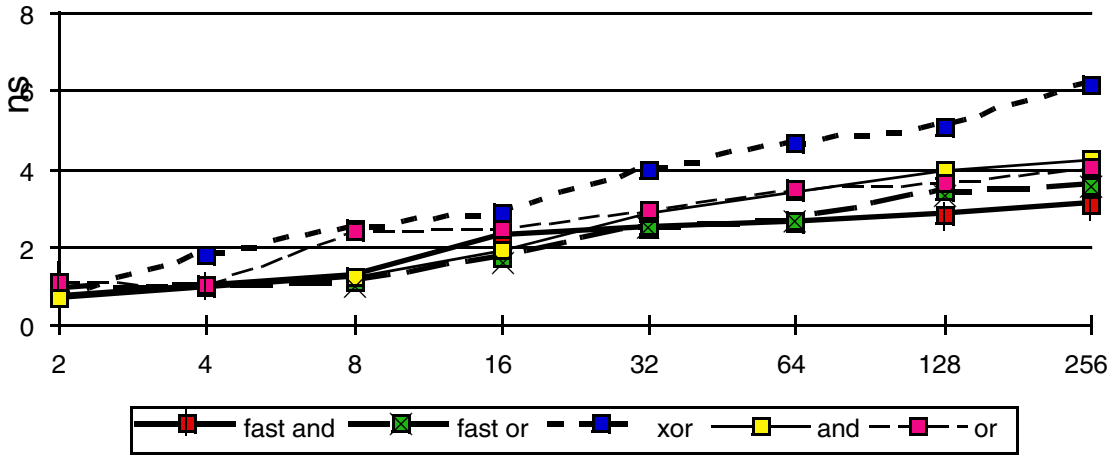
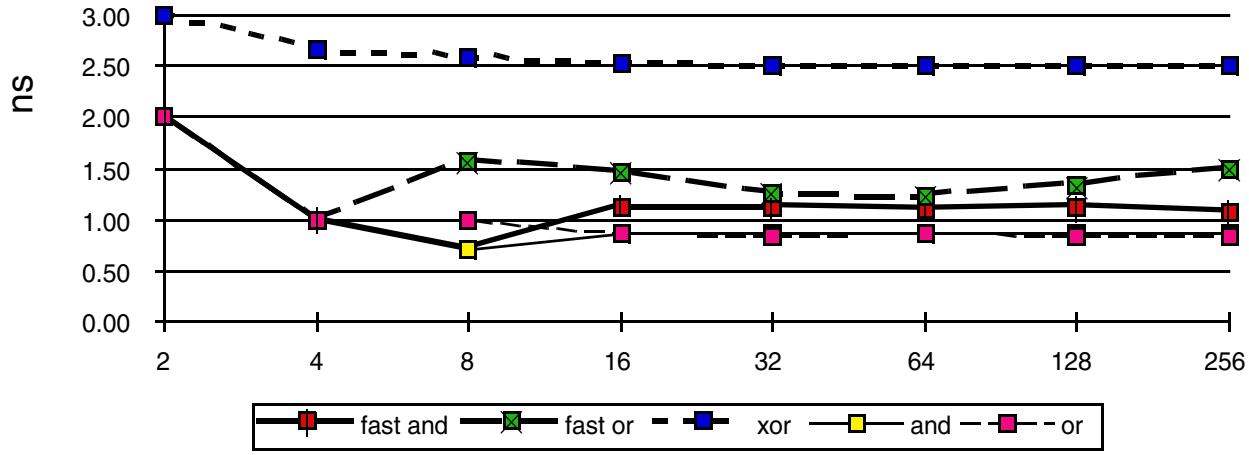


Figure 7-32 AND, OR, and XOR Area per Bit Versus Depth, Normalized by Depth –1



Symmetric Pipelined FIR Filters

To illustrate the effects of complex arithmetic structures and pipelining, the FIR filter structure was chosen. For this example, the data inputs are 6 bits, the coefficient inputs are 10 bits, and the filter has 9 taps. The filter is constructed with a single Wallace tree.

First, consider the case of symmetric (with respect to the data and coefficient inputs) pipelining. The plot in Figure 7-33 shows the final versus desired delay for a range of 40 MHz to 125 MHz. The final delay is generally better than or equal to the desired delay.

Figure 7-33 FIR Filter Final Versus Desired Delay

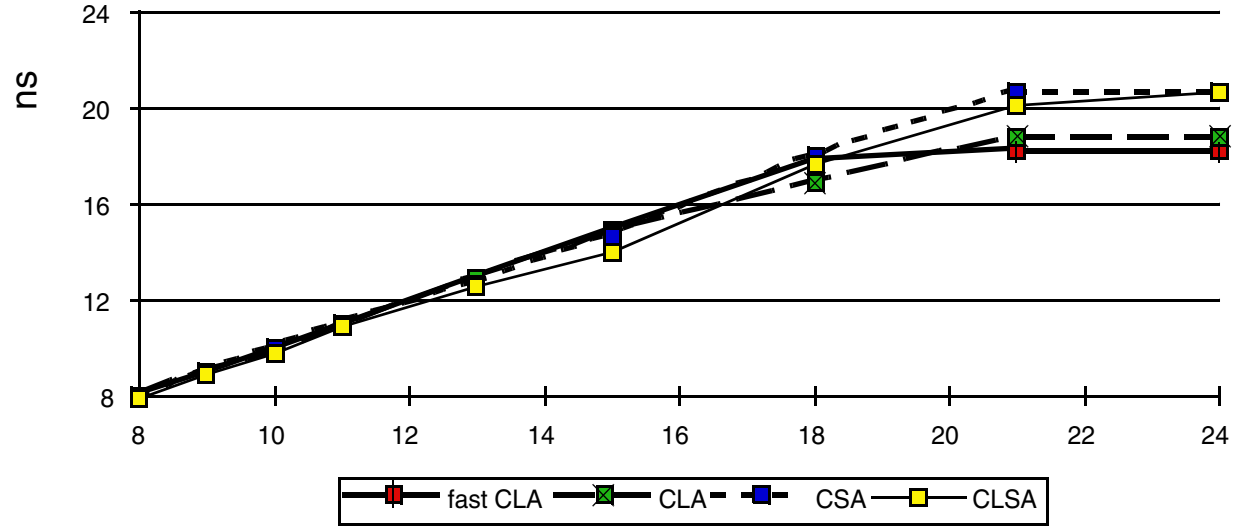


Figure 7-34 shows the effects of the four final adder architectures. The csa provides the best area for a given delay, most likely due to the large delay skews involved and because the csa can be automatically partitioned across pipeline stages.

Figure 7-34 FIR Filter Area Versus Desired Delay

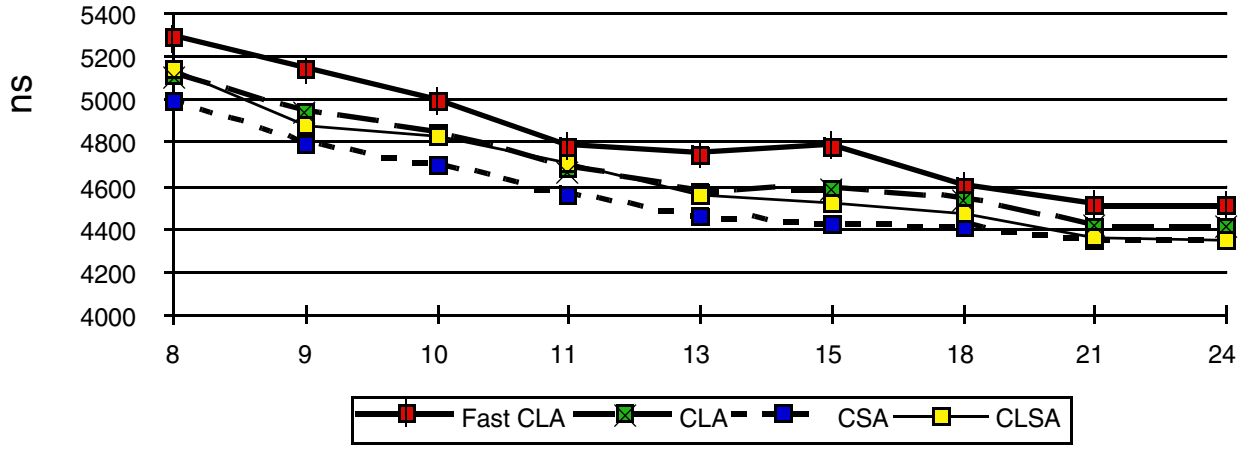
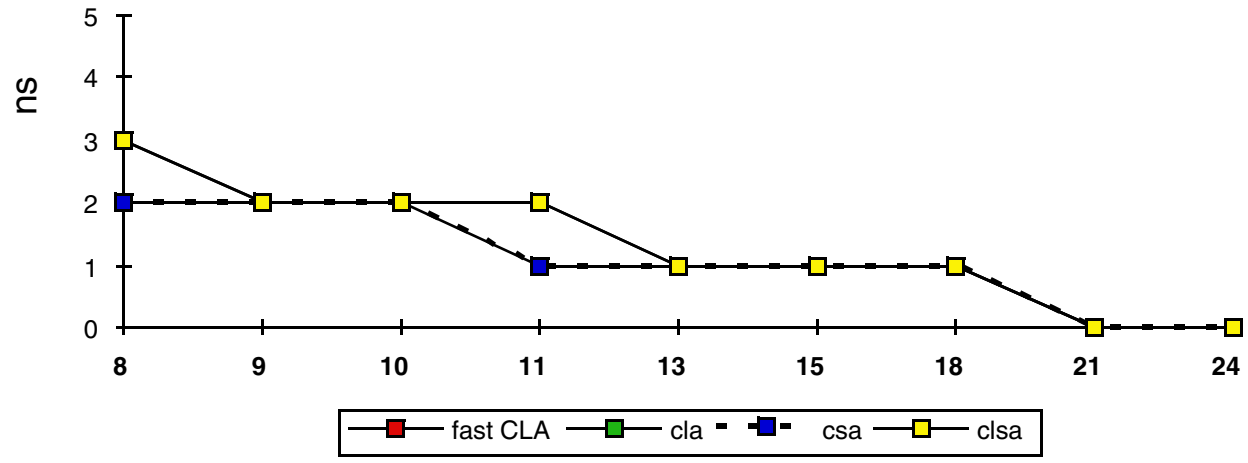


Figure 7-35 shows that the csa (along with the fastcla) provides the best latency. The clsa does not pipeline well and hence suffers from excessive latency.

Figure 7-35 Latency Versus Desired Delay



Asymmetric Pipelined FIR Filters

The `delstate` construct can be employed to implement asymmetric pipelining for latency improvements.

Figure 7-36 shows the delay versus coefficient latency for the same FIR filter shown in Figure 7-35. Note that no data latency is allowed. Figure 7-36 shows that significant “no latency” delay improvement can be obtained. The FIR filter area versus coefficient latency is also shown in Figure 7-37.

Figure 7-36 FIR Filter Delay Versus Coefficient Latency, No Data Latency

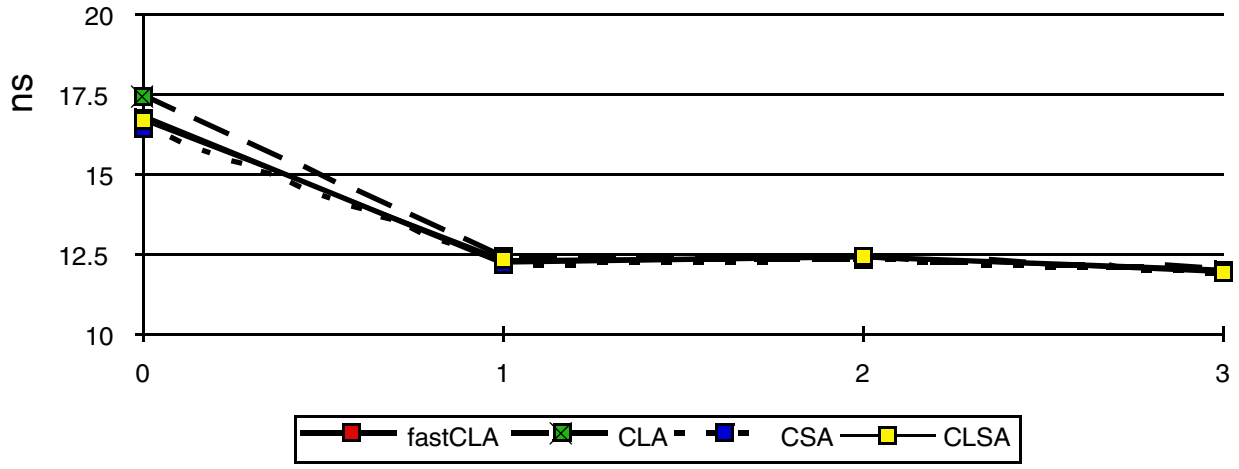
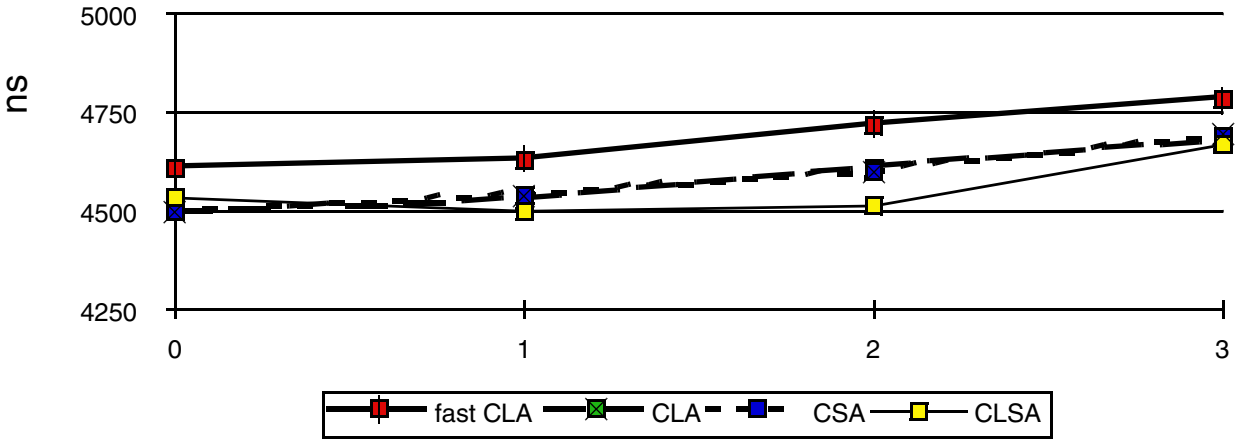
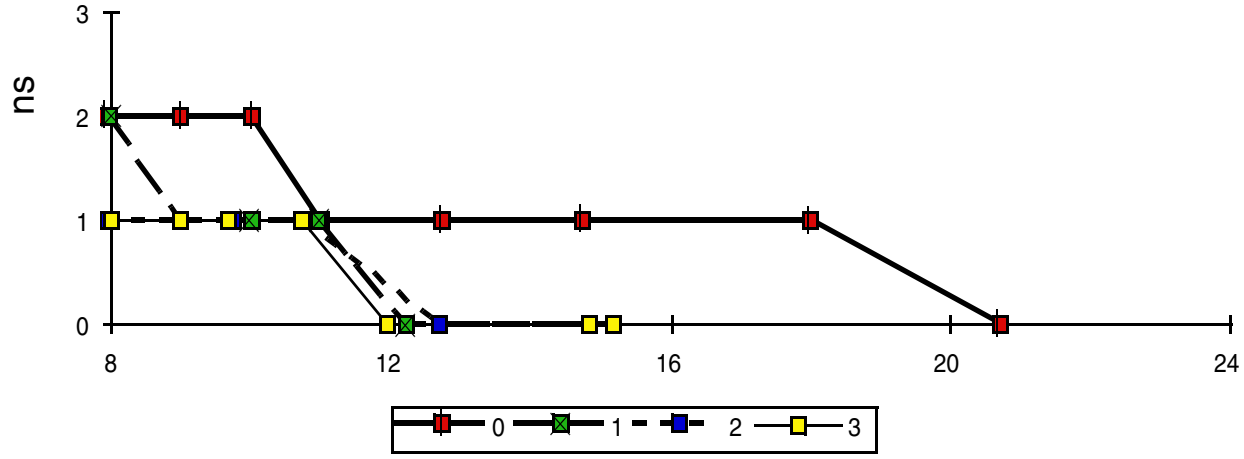


Figure 7-37 FIR Filter Area Versus Coefficient Latency, No Data Latency



To make matters more complex, consider the same case shown earlier in Figure 7-36 and in Figure 7-37, but with additional pipelining allowed. Figure 7-38 shows that asymmetric pipelining can produce latency savings of up to two cycles.

Figure 7-38 Data Latency Versus Actual Delay for Different Coefficient Latencies



Index

Symbols

!=, ==, >=, >, <, <= comparison signal operators 1-42
#define 1-5
#endiff 1-5
#include 1-5
&, |, ^ AND, OR, XOR signal operators 1-44
+, -, and * sum-based signal operators 1-37
<<, >> shift signal operators 1-40
<<<, >>> rotate signal operators 1-40
?: conditional operator 1-47
[ ] bit range brackets and substring brackets 1-3
{ } (substitute) 1-45

A

absolute value integer 2-84
accumulator signal 2-4
adders
  aofcla 2-62, 2-100, 3-10
  cla 3-11
  clsa 3-11
  csa 3-11
  ripple 3-12
  ripple_alt 3-12
addition circuits, building 2-24
alpha blender (graphics) 2-44
AND signal operator 1-44
aofcla adders 2-62, 2-100, 3-10
arguments, number passed to function 2-93
arithmetic logic unit 2-9
attributes
  archopt 3-6
  archtype 3-6
  async_clear 3-7
  async_preset 3-7
  carrysave 3-7
  clock 3-8
  col 3-8
  dcopt 3-8
  delay 3-9
  delstate 3-9
  direxT 3-9
  fadelay 3-10
  fatype 3-10
    aofcla 3-10
    auto 3-10
    cla 3-11
    clsa 3-11
    csa 3-11
    fastcla 3-11
    ripple 3-12
    ripple_alt 3-12
  group 3-12

indelay 3-13
inload 3-13
intround 3-14
list of 3-3
logopt 3-14
maxtreedepth 3-14
modname 3-15
multtype 3-15
multype
  Booth 3-16
  non-Booth 3-16
muxtype 3-17
  andor 3-17
  mux 3-17
  tristate 3-18
outdelay 3-18
outload 3-18
physical 3-19
physicalfunction 3-19
pipeline 3-19
pipeslack 3-20
pipestall 3-20
round 3-20
row 3-21
rpgen 3-21
scan 3-21
selectop 3-22
use_for_size_only 3-22

B

bit range limitations 1-5
bit-width 2-106
blend of two input pixels (graphics) 2-44
buffer 2-15
buffer depth, setting 2-15

C

cells, naming limitation 4-2
cells, required and optional 4-2
cla adders 3-11
clsa adders 3-11
command-line options 5-3
commands, dc_shell
  compile_mcl 5-20
  mcenv 5-20
  read_mcl 5-20
  report_mc 5-20
comparison of two inputs 2-18
comparison signal operators 1-42
compile_mcl command 5-20
concatenation, signal 2-17
condition, preprocessor 1-5
constant, string 2-103
counters, creation of 2-19
CRC encoder/decoder 2-22
critical path
  analysis 2-84
  reporting 2-84
csa adders 3-11
csconvertSelectopOneHot, parser limitation 2-27

D

datapaths, functions that define 2-83
dc_shell commands for Module Compiler 5-20
decoders, creation of 2-30
demultiplexers, creation of 2-31
Design Compiler
  commands 5-20
DesignWare components
  floating-point adder 2-36
  floating-point comparator 2-36
  floating-point converter 2-37
  floating-point divider 2-36
  floating-point multiplier 2-38
  floating-point-to-integer converter 2-37
directive scope
  default 3-2

local 3-2
directives, *(see* attributes)
divide to get quotient and remainder 2-33
divide, using 2-33
don't use cells 3-5, 3-22
DW_add_fp 2-36
DW_cmp_fp 2-36
DW_div_fp 2-36
DW_flt2i_fp 2-37
DW_i2flt_fp 2-37
DW_mult_fp 2-38

E

encoder/decoder, CRC 2-22
environment variables
list of 5-3
relative placement, list of 5-14
reserved, list of 5-17
setting 5-2
error message 2-92
error messages
samples 6-2
even integer 2-96

F

fatal error message 2-92
file, include 1-5
FIR filter 2-38
five-function shifter/rotator (graphics) 2-50
floating-point adder 2-36
floating-point comparator 2-36
floating-point converter 2-37
floating-point divider 2-36
floating-point multiplier 2-38
floating-point signal format
leading 0s 2-65
leading 1s 2-65
floating-point-to-integer converter 2-37
functions
defining datapaths 2-83
hardware 2-2
AccPM 2-6
accum 2-4
alup 2-9
bitrev 2-15
buffer 2-15
cat 2-17
compGE 2-18
count 2-19
crc 2-22
csconvertAddsub 2-24
csconvertSelectopOneHot 2-26
csconvertTruncate 2-27
decode 2-30
demux 2-31
divide 2-33
DW_add_fp 2-36
DW_cmp_fp 2-36
DW_div_fp 2-36
DW_flt2i_fp 2-37
DW_i2flt_fp 2-37
DW_mult_fp 2-38
fir 2-38
gfxBit 2-41
gfxBlend 2-44
gfxLogicop 2-48
gfxShift 2-50
isolate 2-52
join 2-53
LZ 2-54
mac,maccs 2-56
mag 2-59
max2,maxmin,min2 2-61
multp 2-63
norm,norm1 2-65
registers:eqreg, eqreg1, eqreg2, ensreg, preg, sreg 2-69
ResolveLatency 2-73
ResolveLatencyLoop 2-73

sat, sati 2-76
SetLatency 2-77
sgnmult 2-79
shiftlr 2-81
string 2-105
supporting 2-83
abs 2-84
critmode 2-84
critpath 2-84
csconvert 2-88
directive 2-90
disablepath 2-84
enablepath 2-84
error, fatal, warning, info 2-92
fatal 2-92
fnArgs 2-93
formatStr 2-95
hidelat 2-94
info 2-92
is2expN 2-97
is2expNmin1 2-97
isEven 2-96
isOdd 2-96
log2 2-106
max 2-99
min 2-99
param 2-100
showgroup 2-101
string 2-103
string constant 2-103
warning 2-92
width 2-106

G

generate 2-2
generic functions 4-8
cells, required and optional 4-2
graphics
blending input pixels 2-44
merging bit regions 2-41
pixel logic processor 2-48
shifting and rotating 2-50
group information, displaying 2-101

I

include file 1-5
information message 2-92
integer
even 2-96
functions 1-21
less than power of 2 2-97
odd 2-96
operators 1-35
power of 2 2-97
isolation, critical nets from heavy loads 2-52

J

joining signals 2-53

L

language syntax
nonoperators 1-2
operators 1-34
latency equalization register
latency = as given 2-69
latency = maximum of reference list 2-69
latency = sum of reference list 2-69
latency, resolving 2-73
less than power of 2 integer 2-97
linear interpolator (graphics) 2-44

M

maximum and minimum value signal 2-61
maximum value integer 2-99
maximum value signal 2-61
mc -tech command 5-20
mcenv command 5-20

mc.env file 5-2
mcenv utility 5-2
merging bit regions, two inputs (graphics) 2-41
message
  error 2-92
  fatal error 2-92
  information 2-92
  warning 2-92
minimum value integer 2-99
minimum value signal 2-61
multiplexers
  ?: conditional operator 1-47
multiplier-accumulator 2-56
  signal function with carry-save 2-56
multiply plus constant 2-63
multiplying a signal 2-79

N

nets, critical, isolated from heavy loads 2-52
nonoperators
  #define 1-5
  #endif 1-5
  #ifdef 1-5
  #include 1-5
  [ ] bit range brackets and substring brackets 1-3
  endfunction 1-8
  endmodule 1-22
  function 1-8
  function call 1-11
  global 1-15
  if/else 1-15
  input 1-17
  integer 1-19
  integer constant 1-19
  module 1-22
  output 1-25
  repl 1-28
  replicate 1-28
  signed 1-30
  unsigned 1-30
  wire 1-32
numerical comparison of two inputs 2-18

O

odd integer 2-96
operators
  ?: conditional operator 1-47
  { } (substitute) 1-45
  signal
    !=,==,>=,>,<,<= comparison signal operators 1-42
    &,|,^ AND, OR, and XOR signal operators 1-44
    +,-,* sum-based signal operators 1-37
    <<,>>,<<<,>>> shift and rotate signal operators 1-40
    ~ bitwise negation signal operator 1-36
  string 2-105

P

parameter iteration 2-100
paths
  analyzing 2-84
  disabling 2-84
  enabling 2-84
  reporting 2-84
pipeline design to user-defined latency level 2-73
  at end of pipelined code section 2-73
pipeline register 2-69
pixel logic processor (graphics) 2-48
power of 2 integer 2-97
preprocessor
  condition 1-5
  macro, define 1-5
priority encoder 2-55
programmable arithmetic logic unit 2-9

programmable blend of two input pixels (graphics) 2-44

Q

quotient, using divide 2-33

R

read_mcl command 5-20
registers 2-69
relative placement
  environment variables 5-14
  structuring a portion of a design 3-21
remainder, using divide 2-33
report_mc command 5-20
reserved environment variables 5-17
resolving latency 2-73
ripple adders 3-12
ripple_alt adders 3-12
rotate signal operators 1-40
rotator (graphics) 2-50

S

shift left/right 2-81
shift signal operators 1-40
shifter (graphics) 2-50
signal
  accumulator 2-4
  clipping 2-76
  clipping with inverted output 2-76
  compute absolute value 2-59
  concatenation 2-17
  format
    returned as a string 2-95
  inversion 1-36, 3-11
  operators
    !=, ==, >=, >, <, <= comparison signal operators 1-42
    &, |, ^ AND, OR, XOR 1-44
    +, -, and * sum-based signal operators 1-37
    <<, >> shift signal operators 1-40
    <<<, >>> rotate signal operators 1-40
    AND 1-44
    comparison 1-42
    OR 1-44
    rotate 1-40
    shift 1-40
    sum-based 1-37
    XOR 1-44
signal latencies, specifying input 2-77
signals, connected bitwise 2-53
sign-multiplier 2-79
starting Module Compiler 5-20
state registers 2-69
string 2-103
  constant 2-103
  functions 2-105
  operators 2-105
  variables 2-103
string constant 2-103
subtraction circuits, building 2-24
sum-based signal operators 1-37

T

truncation, performing 2-27

V

variables string 2-105

W

warning messages 2-92

X

XOR signal operator 1-44